Improving Cache Performance Via Active Management

by

Edward S. Tam

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
1999

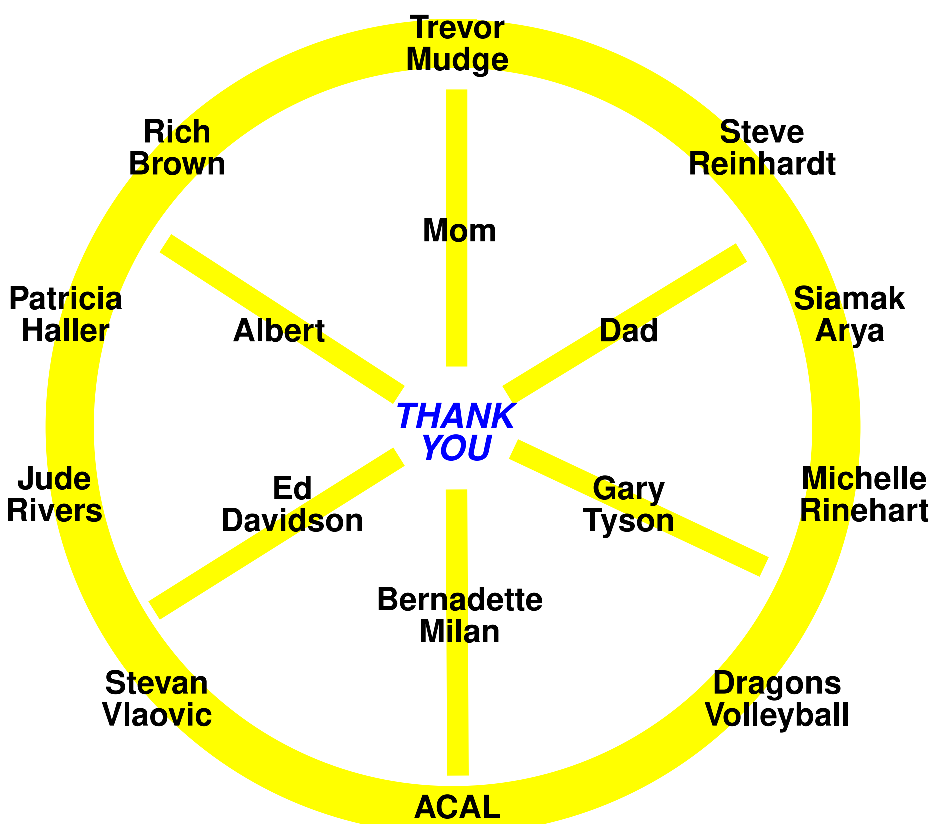
Doctoral Committee:

Professor Edward S. Davidson, Co-Chair
Assistant Professor Gary S. Tyson, Co-Chair
Professor Richard B. Brown
Professor Trevor N. Mudge
Assistant Professor Steven K. Reinhardt

To Mom and Dad, for everything.

ACKNOWLEDGEMENTS

Many people have made significant impacts in my life throughout my graduate school career. While I would be wont for space to individually thank each of the people who have touched my life during this time, I would like to express my heartfelt thanks to the following people for their great love, support, encouragement, and belief in me throughout:



ABSTRACT

IMPROVING CACHE PERFORMANCE VIA ACTIVE MANAGEMENT

by

Edward S. Tam

Chairs: Edward S. Davidson, Gary S. Tyson

This dissertation analyzes a way to improve cache performance via active management of a target cache space. As microprocessor speeds continue to grow faster than memory subsystem speeds, minimizing the average data access time grows in importance. As current data caches are often poorly and inefficiently managed, a good management technique can improve the average data access time. Cache management involves two main processes: block allocation decisions and block replacement decisions. Active block allocation can be performed most efficiently in multilateral caches (several distinct data stores with disjoint contents placed in parallel within L1), where blocks exhibiting particular characteristics can be placed in the appropriate store.

To aid in our evaluation of different active block management schemes, we have developed a multilateral cache simulator, *mlcache*, which provides a platform whereby different cache schemes can easily be specified, and produces evaluation statistics that can help explain their performance. Using *mlcache*, we have been able to evaluate the performance of proposed multilateral cache schemes and to derive new, better performing schemes. Our results show that multilateral schemes outperform traditional caches of similar size and often rival the performance of traditional caches nearly twice as large. However, the performance difference between previously-proposed implementable schemes and a multilateral configuration that uses a non-implementable near-optimal replacement policy is large. This disparity is due mainly to the simple prediction strategies presently used in the implementable schemes, along with their limited management of blocks while resident in the L1 cache structure.

We introduce a new multilateral allocation scheme, Allocation By Conflict (ABC), which outperforms all previously proposed reuse-based multilateral configurations and performs comparably to multilateral schemes that have significantly more hardware requirements (particularly Victim, which requires a data path between its A and B caches). The ABC scheme incurs the lowest hardware cost of any of the proposed multilateral schemes, yet it performs the highest and is the most easily implementable. The ABC scheme requires the addition of only a single additional bit per block in cache A and a very simple logic circuit for making the allocation decisions. The ABC scheme's performance advantage also scales well as the sizes of the caches are increased and as the associativity of the A cache is increased.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 Addressing the processor-memory speed gap

Minimizing the *average data access time* is of paramount importance when designing high-performance machines. Unfortunately, access time to off-chip memory (measured in processor clock cycles) has increased dramatically as the disparity between main memory access times and processor clock speeds widens. The effect of this disparity is further compounded as multiple-issue processors continue to increase the number of instructions that can be issued each cycle. As the average data access time becomes much larger relative to processor cycle time, further improvements in microprocessor architecture bring limited gains, as the processor will more and more often be waiting for requested data to return from memory. Even if aggressive compiler or hardware optimizations are utilized to execute independent instructions in parallel, inevitably a data access will stall one or more paths of execution, potentially stalling the processor as a whole until requested data is returned. In order to maximize the benefits attained by architectural improvements to microprocessor designs, the average data access time must be minimized by significantly increasing the hit ratios to the faster levels of the memory hierarchy. Given the rate at which memory access times are increasing relative to clock cycle times [1], masking the access latencies has become ever more critical to overall system performance.

While there are many approaches to minimizing the average data access time, today's most common solution is to incorporate multiple levels of cache memory on-chip, but still allocate and replace their blocks in a manner that is essentially the same as when caches first appeared three decades ago. Recent studies [2][3][4][5][6] have indicated that there may be better ways to configure and manage the first-level (L1) cache. *Active cache management*, which consists of active block allocation and replacement, can improve the performance of a given size cache. Active management uses information, such as a block's reference behavior in past and present tours[1], to make block allocation and replacement decisions; this is opposed to the passive management schemes traditionally used in caches, in which blocks are allocated in a cache structure via a simple modulo mapping of their addresses and replaced according to static policies. More specifically, active block allocation exploits reference behavior and places blocks in the cache so as to maximize their usefulness and reduce any harm that their allocation may impart to the cache state. Active block replacement attempts to identify the most useful blocks in the cache using information such as reference behavior in past and present tours, and retain them in the cache in preference to blocks deemed to be less useful.

In order to accommodate varying block usage within a given cache structure, several schemes, NTS [3], MAT [4], Dual [5], and PCS [6], have been proposed that incorporate an additional data store within the L1 cache structure to increase allocation and replacement flexibility in the resulting *multilateral* [7] cache so that reuse pattern information can be better exploited. We use the term *multilateral* to refer to a level of

1. A block tour starts when a block is fetched to cache from memory and ends when the block is evicted.

cache that contains two or more data stores that are accessed in parallel and have disjoint contents. We refer to the larger of the data stores as the A cache, and the smaller as the B cache. The previously proposed schemes perform active block allocation, though they do not perform active replacement; each scheme instead relegates block replacement decisions to a simple hardware replacement algorithm. It has been shown that multilateral designs perform as well as or better than larger, single data store caches while requiring less die area [7]. For a given die size, reducing the die requirements to attain a given rate of data supply can allow other resources to use the saved space, e.g. incorporating higher levels of memory on chip and/or dedicating more space to branch prediction, data forwarding, instruction supply, the instruction reorder buffer, and finally functional units and instruction issue width.

Many other techniques for reducing or tolerating the average data access time have been employed or proposed. Prominent among these are: 1) store buffers, used to reduce bus contention by delaying writes until idle bus cycles occur; 2) non-blocking caches, which overlap multiple load misses while fulfilling other requests that hit in the cache [8][9]; 3) hardware and software prefetching methodologies that attempt to preload data from memory to the cache before it is needed [10][11][12][13][14]; and 4) XOR-mapping functions [15][16] and column-associative caches [17], which reduce the effect of conflicts in cache block allocation. While these schemes do contribute to reducing or tolerating average data access time, this dissertation approaches the access latency problem from the premise that the average data access time can be reduced by exploiting data usage information to actively manage the state of a multilateral L1 cache. This approach can be used in conjunction with these other techniques to reduce the average data access time further.

1.2 Contributions of this work

Our approach to improving cache performance, and thus reducing average data access time via active cache management, has resulted in two main contributions. The first is the development and deployment of a flexible, timing-sensitive multilateral cache simulator, *mlcache*, which allows one to easily model and compare the performance of many different multilateral cache designs on a common platform, allowing one to assess each scheme's performance based on the management decisions made, for comparably-sized underlying configurations. The *mlcache* simulator provides detailed outputs to support in-depth analysis of each scheme's performance. These include simple hit/miss ratios, delayed hit ratios, average access latencies, and reuse behavior, among others, which are useful metrics that can help identify strong and weak points of an evaluated cache scheme. Using *mlcache*, we have been able to evaluate the performance of previously proposed multilateral cache schemes, and to derive new, better performing schemes.

The second contribution of this dissertation is the introduction of a new multilateral allocation scheme, Allocation By Conflict (ABC), which outperforms all previously proposed reuse-based multilateral configurations and performs comparably to multilateral schemes with significantly more hardware requirements (particularly Victim [18], which requires a data path between the A and B caches). The ABC scheme incurs the lowest hardware cost of any of the proposed multilateral schemes, yet it performs the highest and is the most readily implementable. The ABC scheme requires only a single additional bit per block in the cache A directory and a very simple logic circuit for making the allocation decisions. The ABC scheme's performance advantage also scales well as the sizes of the caches are increased and as the associativity of the A cache is increased.

While the ABC scheme is easily realized and performs very well, performance can undoubtedly be improved further (e.g. by the incorporation of active replacement, prefetching, compiler hints for allocation, XOR-mapping, etc.) The flexibility, modularity, and openness of the *mlcache* simulator allows any of these additions to be easily incorporated. These additional features may in turn drive the development of even newer and better active cache management schemes whose goal would be to further reduce the average data access time beyond what the ABC scheme has already achieved.

1.3 Dissertation overview

Throughout the dissertation, prior work will be discussed and evaluated in each chapter as appropriate.

In Chapter II we present *mlcache*, a flexible multilateral cache simulation tool, which we developed and used for our evaluation and exploration of multilateral cache designs. *mlcache* is an event-driven, timing-sensitive cache simulator based on the Latency Effects cache timing model [19]. It can be easily configured to model various multilateral cache configurations by using its library of cache state and data movement routines. For interactions not modeled in the library routines, users can write their own management routines and call them from this simulator which is structured for easy extensibility. The tool can be used in a stand-alone mode or joined to any of a wide range of event-driven processor simulators; it has already been used in conjunction with *RCM_brisc* [20] and *SimpleScalar* [21]. A combined processor-and-cache simulator, such as *SimpleScalar+mlcache,* can provide detailed evaluations of multiple cache designs running target workloads on proposed processor/cache configurations. We show how a wide range of proposed multilateral cache schemes can easily be modeled using *mlcache,* and present performance analyses based on outputs obtained from the simulator.

In Chapter III, we explore the benefits of using reuse information for past block tours in making allocation decisions for future block tours. We compare three multilateral cache schemes: MAT [4], NTS [3], and PCS [6], and the results of their exploitation of reuse information. We configure their caches to be as similar as possible and evaluate them using the *mlcache* cache simulator connected to the *sim-outorder* processor simulator from the SimpleScalar toolset. We find that making placement decisions based on effective address-based block reuse, as in the NTS scheme, outperforms the macroblock-based and PC-based approaches of MAT and PCS, respectively. Nevertheless, all three schemes perform comparably to larger direct-mapped caches and better than associative caches of similar size.

We then examine the performance of optimal and near-optimal multilateral caches to determine the performance potential of multilateral schemes. Optimal and near-optimal schemes excel in block replacement decisions, while each block allocation decision is a direct consequence of the corresponding replacement decision. We compare the performance of two implemented multilateral schemes to one another and to the near-optimal scheme in order to determine the underlying reasons for their levels of performance. This analysis shows that the implemented schemes might perform better by improving their block allocation choices and/or by incorporating more robust replacement mechanisms.

In Chapter IV, we explore the performance of a new multilateral allocation scheme, designed to address the shortcomings of the schemes evaluated in Chapter III.

The Allocation By Conflict (ABC) scheme is primarily based on reducing conflicts by using a multilateral cache. It makes allocation decisions based on recent (rather than past tour) usage of data in the cache; furthermore, it focuses on the current tour behavior of the block that is a candidate for replacement, rather than the past tour behavior of the incoming miss block. A Random allocation scheme is also evaluated as a performance baseline for multilateral caches of a given size and configuration.

We compare the ABC and Random allocation schemes to NTS and PCS, the best among previous reuse-based allocation schemes, and to the Victim scheme [18], previously found to perform best when the A cache is direct-mapped. We find that the ABC scheme outperforms the reuse-based allocation strategies, and even performs comparably to Victim when the A cache associativity is low. Interestingly, Random performs well with low A cache associativity also. As the associativity of the A cache increases, the reuse-based schemes (NTS, PCS, and ABC) all continue to perform well, while the performance benefits of Random and Victim degrade. The ABC scheme is the most easily implementable multilateral cache scheme of all those evaluated, requiring the least hardware (only a single additional bit per block in the cache A directory and very simple decision logic), yet its performance is the best overall.

In Chapter V we present our conclusions and discuss some future directions.

CHAPTER II

MLCACHE: A TOOL FOR EVALUATING THE PERFORMANCE OF ACTIVE CACHE MANAGEMENT SCHEMES

2.1 Introduction

The *mlcache* multilateral cache simulator was developed to help designers early in the design cycle decide what cache configurations might best help attain the performance goals of a target processor. Early in the design cycle, metrics such as miss ratio and bandwidth requirements can be used to narrow the spectrum of caches considered for a processor design. Before detailed cycle-by-cycle simulators are constructed for final candidate designs, it is often very helpful to use higher-level timing simulators to obtain a more detailed evaluation of cache and processor performance.

In this chapter we present the *mlcache* tool and show how it can be used to compare the performance of several different L1 data cache configurations when running the SPEC95 benchmarks. While multilateral designs can also be used for instruction caches, the lower predictability of references to data is more suited to exposing the benefit of multilateral cache designs. The multilateral cache configurations that we consider in this evaluation include the Assist cache [22][23], Victim cache, NTS cache, and PCS cache. We show how each of these can be easily realized using the *mlcache* tool, and expose the performance of each scheme for a set of cache parameters. Though *mlcache* is designed for multilateral cache configurations, it can also handle one or two levels of single structure caches. We use *mlcache* to compare the performance of the multilateral caches with single structure caches of increased size.

2.2 Cache tours and their classification

The effectiveness of a cache management scheme can be measured by its ability to minimize the number of block tours that are accumulated during a program run. Individual cache block tours can be monitored in *mlcache* and classified based on their reuse pattern. A tour that sees a reuse of any word of the block is considered dynamic temporal; a tour that sees no word reuse is dynamic nontemporal. Both dynamic temporal and dynamic nontemporal tours can be further classified as either spatial, when more than one word was used, or nonspatial. This allows us to classify all tours into four groups: 1) nontemporal nonspatial (NTNS), 2) nontemporal spatial (NTS), 3) temporal nonspatial (TNS), and 4) temporal spatial (TS). We will refer to these tour types when defining some of the reuse-based multilateral schemes.

2.3 Modeling and simulating multilateral caches

In order to model and compare multilateral cache configurations, it is helpful to have a figure from which each of the configurations can be derived. Figure 2.1i shows a "fully-connected" processor-memory system with two cache structures in L1 backed by a main memory. Depending upon the specific configuration being evaluated, some of the paths will be deleted. The direct path between memory and processor is not included in

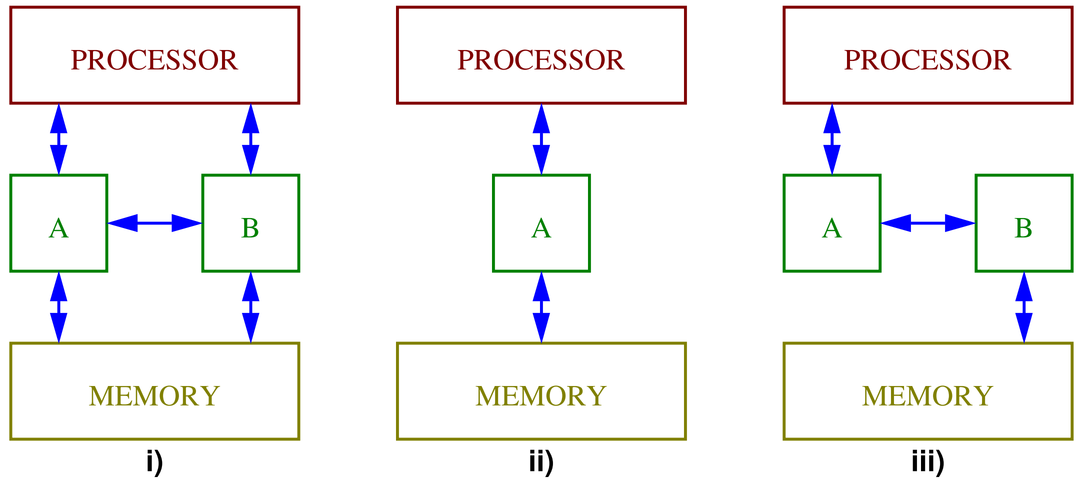


Figure 2.1: Schematic views of processor-memory systems with two caches: i) fully-connected, ii) a single structure cache, and iii) a two-level cache.

the figure, as it is assumed that even cache-bypass data that returns to the processor directly from memory must still go through the cache unit. The effects of a memory to processor transfer can be obtained by assigning it appropriate parameter values for traversing the corresponding memory-to-cache and cache-to-processor paths.

This figure can thus represent, at a high level, practically any system consisting of two cache structures, a processor, and memory. (Note that the "memory" in this figure can actually represent a second level cache. In explaining this model, however, we will assume that the level of memory backing the two caches is the system's main memory.) By removing particular arcs and elements from Figure 2.1i, different cache configurations can be represented, e.g. a traditional single structure cache backed by main memory (Figure 2.1ii) or a 2-level cache, where B is the L2 cache and A is the L1 cache (Figure 2.1iii). The A and B caches can have different sizes, associativities, replacement policies, etc., which are specified to *mlcache* by simply assigning parameter values.

Figure 2.2 shows representations of several multilateral cache configurations. Figure 2.2i shows the Assist cache configuration, used in the HP PA-7200 [22][23]. All

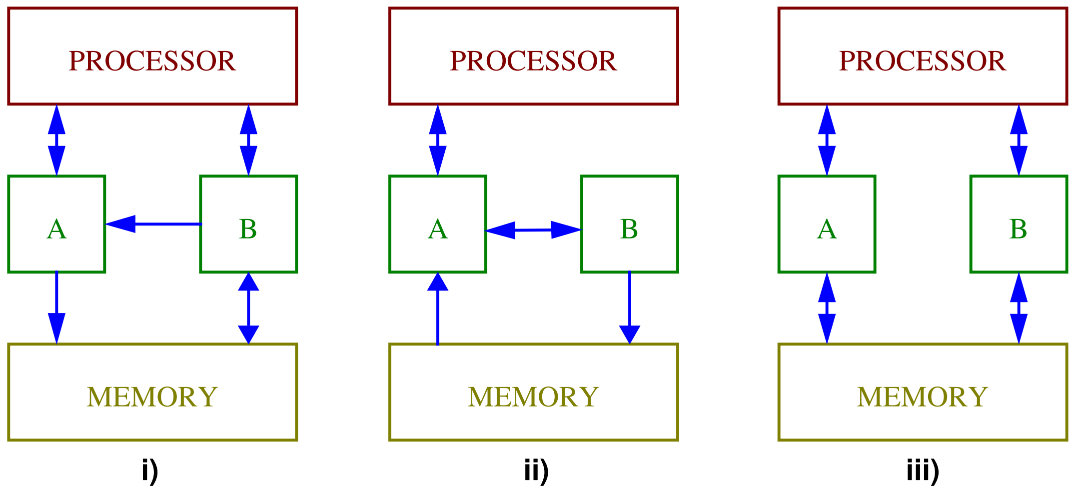


Figure 2.2: Representations of some multilateral cache configurations: i) Assist cache, ii) Victim cache, and iii) NTS and PCS caches.

blocks that enter the cache from memory must enter through the Assist buffer (the B cache). Note that there is no direct memory-to-A cache transfer path in Figure 2.2i. The B cache uses FIFO replacement — blocks evicted from the B cache are moved ("promoted") into the A cache unless they are tagged with a compiler-supplied spatial-only hint. Once a block has been promoted to the A cache, it resides there until it is replaced under an LRU policy. In the basic Assist implementation, dirty blocks that are replaced in A are written back to main memory; thus, there is no direct path from the A cache to the B cache. Also, blocks tagged spatial-only in the B cache are written back to main memory if they are dirty when replaced. Further details of Assist cache operation are presented in Section 2.6.

In the Victim cache (Figure 2.2ii), blocks replaced in A move to B, and move back to A if hit while in B. In the NTS or PCS cache (Figure 2.2iii), blocks deemed to be temporal are allocated to A, and those deemed nontemporal to B. Detailed descriptions of the Victim, NTS, and PCS caches are presented in Section 2.6.

Early in the design cycle, it is helpful to incorporate latency effects when considering memory accesses, as miss ratios alone are not a sufficiently accurate performance metric of a target cache design. When cycle-level simulation is performed, two cache designs with similar miss ratios can have markedly different program execution times depending upon the actual latency of each access. Thus, we use the Latency Effects (LE) cache timing model [19], described below, to account for the latencies seen by each memory access. The LE cache model is a paramaterizeable model that determines the latency for each access by considering leading and trailing edge effects, bus width and contention effects, port conflicts, and the number of accesses in flight (NOA) that the cache allows before blocking. For single structure caches, the LE model is used to determine the latency of hits, misses, and delayed hits to the cache. Delayed hits are accesses to data currently returning to the cache on behalf of earlier misses to the same block. Delayed hits incur latencies greater than cache hits, but typically less than cache misses.

For multiple-cache configurations, computing the latency of an access is more complex. A latency is assigned to each of the relevant links in the figure for each type of operation to be performed. For instance, if an access is made to a block during its promotion from B to A, the promotion time is included in the access latency. The latency of an access can be determined by summing the time to traverse each link of the path from where the block resides at the time of the request to its final destination in the processor. For example, in the Assist cache configuration, the nominal miss latency, trailing-edge effects, and bus width and contention considerations are incorporated in the memory-to-cache path, while the latency between caches and related trailing-edge effects (if the inter-cache latency is significant) are included in the cache-to-cache path. Regardless of the cache configuration, each access is subject to the added latencies, if any, due to port conflicts and NOA.

2.4 Latency Effects (LE) cache timing model

The performance of a particular cache configuration depends not only on the miss ratio incurred during the execution of a particular workload but also on where in the program's execution the misses occur and the latency of each miss, which taken together indicate the number of stall cycles[2] due to each miss. However, useful timing simulation of caches is typically unavailable until late in the design stage. Using today's behavioral simulators, simple, traditional caches are evaluated early in the design cycle; however, novel cache designs are often not considered since they are difficult to model.

The issue of providing more useful cache timing simulation analysis early in the design cycle has been addressed by the Latency Effects (LE) cache model [19], which incorporates latency-adding effects into a behavioral-level simulation, particularly trailing-edge effects, bus width considerations, the effects of bus contention and port conflicts, and the number of outstanding accesses that a cache can handle before blocking.

2.4.1 The LE cache model

The LE model works as follows: First, consider a single cache with a perfect backing memory. When the processor makes an access while the cache is "quiet," i.e. there is no activity still in process for any earlier access, the cache state is interrogated and this access is assigned a nominal latency depending on whether it is a load or store, and a hit or miss.

An access is said to commence in the cycle in which it is accepted by the cache and

---

2. In in-order issue machines, stall cycles refer to the number of cycles during which the issue unit is stalled waiting for some previously issued instruction to receive its data from memory and complete execution. In out-of-order issue machines, stall cycles refer to the number of cycles that dependent instructions must wait for desired data to return from memory. However, these stall cycles may not directly impact overall execution time if other instructions are able to issue and/or execute while the memory request is being satisfied.

a read access is said to complete in the cycle in which data is returned to the processor (even though some activity related to finishing this access may still be pending in the cache). A read access to a quiet cache thus commences at (the beginning of) the same cycle, X, in which the processor makes the access request and completes at (the end of) cycle $X+L$, where L is the nominal latency for this access. In this model, a single-cycle cache (in which a read hit access to a quiet cache returns data to the processor at the end of the same cycle in which the access request is made) is assigned a hit latency of 0, since any reasonable processor model is already accounting for that single cache access cycle.

In general, activity of prior accesses that are still in progress when an access is made may delay the commencement of an access or increase the latency of the access beyond its nominal value due to other “latency-adding” effects. The LE model considers the effects of trailing edges, memory bus width and availability, the number of outstanding accesses allowed before the cache blocks, and port conflicts.

Trailing-edge effects may increase the latency of accesses made to cache blocks that are currently moving to the cache from the next level of memory as the result of a previous miss. For example, if an access is made to block B at cycle Y, but a previous access to block B was made at time X and assigned a latency (including latency-adding effects) of L, then this access can complete no earlier than cycle $X+L$, and its latency is increased, if necessary, to insure this. Since such requests may not incur the full miss penalty and since they do not cause any additional traffic between the cache and the memory, they are generally reported as hits by cache simulators. However, since due to this trailing-edge effect they generally experience a latency that is greater than the nominal hit latency, the LE model reports them separately as *delayed hits*.

Furthermore, if the bus between the cache and the next level of memory is not wide enough to return a full cache block in one cycle or if the bus is busy satisfying an earlier miss request, then the latency is further increased by one or more cycles. Trailing-edge effects, including these bus considerations, may thus increase the minimum completion time for this reference as a function of when the data for this access will be made available by a previous miss.

Another factor that influences the completion time of a memory access is the number of outstanding accesses (NOA) that a cache can handle before it blocks. Once the cache is blocked, no new accesses can commence until at least one of the outstanding accesses completes. Typically, caches allow “hits under misses,” i.e. hits can be completed while the cache is still serving a previous miss; many caches also allow multiple misses to be outstanding. The LE model uses NOA to model this feature. A blocking cache has NOA=1; a fully nonblocking cache has infinite NOA (the cache never blocks regardless of the number of accesses in flight). If the number of outstanding accesses equals NOA when an access is made, then the commencement of this access is delayed until one of the outstanding accesses is fully served, and no other access may be made until the cycle in which this access commences. If the previously assigned access completion time is less than this commencement time plus the nominal latency, this difference is then added to the completion time.

Finally, port conflicts are considered. If no appropriate port to the processor is available at the current completion time of this access, then the completion time is further increased to the first cycle at which such a port is available.

The Latency Effects (LE) cache model thus accounts for nominal hit and miss times, plus the added delays due to each of the aforementioned effects of prior accesses on the timing of an access. Some timing effects are not accounted for in the model, including

cache size	associativity	block size	word size
number of read ports	number of write ports	number of read/ write ports	NOA
CPU-to-cache bus width	cache-to-memory bus width	replacement policy	return policy (requested word first/ first subblock first)
hit latency	read miss latency	write miss latency	

Table 2.1: A listing of parameters for caches modeled in the *mlcache* simulator. The parameters are read in from an input file and can be changed for each simulation run without recompiling the simulator.

the effects of write-through vs. write-back caches, write-back buffers, and the time required to obtain the system bus in the presence of other users. The effects of TLB misses, page faults, etc. are also not considered. These latency-adding effects may be incorporated in future refinements of the LE model if needed to improve the accuracy of the LE model.

2.4.2 Implementing the LE cache model

The LE cache model was implemented on top of an existing behavioral level simulator, *DineroIII* [24], via a delayed update mechanism. Delayed update is used to allow a behavioral cache simulator to account for latency-adding effects. The use of delayed update causes the effects of an access, i.e. an access' placement into the cache, the removal of the replaced block, etc., to occur only after the calculated latency of the access has passed. The resulting simulator handles a wide variety of cache configurations, as determined by the inputs to the simulator. A list of the inputs handled by the LE cache simulator is shown in Table 2.1.

In addition to typical behavioral cache simulator outputs such as the hit and miss ratios, number of loads and stores, etc., the LE cache simulator also provides delayed hit information, the average access latency for loads and stores seen during the execution of the program, and the cycles due to each of the latency-adding effects that the model incorporates. Each of these statistics can be used by a computer designer or programmer to improve the performance of a given program on a particular machine. For instance, if a significant number of cycles due to port conflicts is seen during the execution of target programs, a designer might increase the number of ports to the cache (via true multi-porting or use of a more cost-effective Locality-Based Interleaved Cache [25]) to reduce the number of port conflicts, thereby decreasing execution time and increasing the machine's performance. As another example, knowledge of trailing edge accesses can help the programmer or compiler designer lay out data and schedule memory accesses so as to avoid these additional delays during program execution [26].

2.5 *mlcache* — an easily configurable tool

2.5.1 High-level parameterization

To make *mlcache* easily retargetable, we chose to provide a library of routines that a user could choose from when deciding what actions take place in the cache at a given time. The routines are accessed from a single C file named `config.c`. The user simply modifies `config.c` to describe all of the desired interactions shown in Figure 2.1 between the caches, processor, and memory. The user also controls when the actions occur via the delayed update mechanism built into the cache simulator. Table 2.2 shows the routines provided for the user to choose from and a brief description of each. If more interactions are needed than those provided, they can then be coded into the simulator by hand — examples of such user-added routines are presented below with the multilateral

Support Routine	Description
check_for_cache_hit()	check to see if an accessed block is present in the cache
update()	place an accessed block into the cache
move_over()	move an accessed block from one cache to another
do_swap()	move an accessed block from cache1 to cache2 and move the evicted block to cache1
do_swap_with_inclusion()	place an accessed block into both cache1 and cache2 and move the evicted block from cache2 to cache1
do_save_evicted()	move the block evicted from cache1 to cache2
find_and_remove()	remove a block from a cache
check_for_reuse()	determine if a block exhibits temporal behavior

Table 2.2: The basic support routines provided with the *mlcache* simulator. The user can call these routines from a configuration file (config.c) to control the cache state and interactions.

cache implementations. However, the routines that are provided are already sufficient to model many multilateral cache designs.

While *mlcache* models many of the effects seen by a memory access in a multilateral configuration, some key effects are still not accounted for. Multilateral configurations that incorporate prefetching, e.g. with a streaming buffer [18], cannot be dealt with because hardware prefetching has not been included in the current implementation. Also, some configurations, e.g. a smaller or less associative cache "backing" a larger or more associative (possibly multilateral) cache, can potentially violate the multi-level inclusion principle [27]; the potential for this violation is common in multilateral caches and has not been addressed in our current studies.

Figure 2.3 shows portions of the `config.c` file that models an Assist cache configuration using the provided routines. As can be seen, the operations in the `config.c` file are all very high level and easily understandable, thereby relieving the user from learning the intricacies of the cache simulator's low-level operations in order to

```
/* this is the standard handler for each access. it checks in the A cache
   to see if the data is there first. if it isn't, it checks in the B cache.
   if it's present in a cache, it handles the appropriate cache hit.
   if the access misses in both caches, a miss is processed.
   other designs may not need both caches checked (e.g. multilateral cache
   schemes that partition the memory access stream based on some criteria
   like address (odd/even), functionality (integer/floating point), etc.). */

long long handle_access(long long cycle_count, UpdateEntry *Entry) {
 /* check for hit in A cache "first" */
 if(!check_for_cache_hit(cycle_count,Entry))
    /* miss in A cache-check in B cache */
    if(!check_for_cache_hit(cycle_count,Entry))
      /* miss in both - handle the miss */
      access_time = handle_miss(cycle_count, Entry, Conflict_Entry);
    else /* hit in B cache (after miss in A cache) - handle the B cache hit */
      access_time = handle_B_cache_hit(cycle_count, Entry);
 else /* hit in A cache - handle the A cache hit */
    access_time = handle_A_cache_hit(cycle_count, Entry);
 return access_time; }

long long handle_A_cache_hit(long long cycle_count, UpdateEntry *Entry) {
  /* hit in A cache, so just update stack, etc. for acache */
  Entry->on_completion = DO_UPDATE;
  Entry->access_latency = (long long)cache_latency;
  Entry->which_cache = ACACHE;
  return(handle_hit_timing(cycle_count,Entry,&(Entry->A))); }

long long handle_B_cache_hit(long long cycle_count, UpdateEntry *Entry) {
  /* hit in B cache, so just update stack, etc. for acache */
  Entry->on_completion = DO_UPDATE;
  Entry->access_latency = (long long)cache_latency;
  Entry->which_cache = BCACHE;
  return(handle_hit_timing(cycle_count,Entry,&(Entry->B))); }

long long handle_miss(long long cycle_count, UpdateEntry *Entry,
                      UpdateEntry *Conflict_Entry) {
  /* for assist cache, if something falls out of the B cache on the
     update, it is placed in the A cache in a FIFO fashion */
  if(Entry->B.dap.accesstype == 0)
    Entry->access_latency = (long long)main_mem_latency_r;
  else
    Entry->access_latency = (long long)main_mem_latency_w;
  Entry->which_cache = BCACHE;
  Entry->on_completion = DO_SAVE_EVICT;
  Entry->move_direction = B_TO_A;
  return(handle_miss_timing(cycle_count, Entry,&(Entry->B))); }
```

Figure 2.3: Part of the `config.c` file used to implement the Assist Cache. Setting on_completion to DO_SAVE_EVICT causes the item evicted from the B cache on a miss to be moved to the A cache (dictated by move_direction being set to B_TO_A), as required.

model a new cache.

2.5.2 Assessing latencies for multiple caches

Accounting for latencies between caches is a simple extension of the LE cache model — given that we know what operation is occurring, we can add the corresponding latency onto the access time and then adjust for any latency-adding effects. For this study, we assume that dedicated busses (as wide as the smaller cache's block size) are present between the caches (if required by a scheme) so that we may ignore bus width considerations between the caches for moves between A and B. We also assume that, given these dedicated busses, there are dedicated ports for accesses traveling between the caches (e.g. this permits a processor read from the A cache while a different element is being moved to A from B). Implementations of these caches in a real, well-designed machine would likely satisfy these assumptions.

Different latencies can also be assigned to a path depending upon the operation that is being performed. The latency assigned for the `move_time` can differ among the cache configurations, as shown in Table 2.3. For an Assist cache (Figure 2.2i), moves between the caches are always in the direction from B (buffer) to A (main cache). Thus, in our experiments, a move in the Assist cache configuration requires a single cycle, meaning that an access that hits a block being promoted from the B cache to the A cache is satisfied with a two cycle latency (one cycle for the move and one cycle to return the data to the processor). Accesses that hit in the A cache are returned in the next cycle, as are accesses that hit in the B cache.

For a Victim cache (Figure 2.2ii), promotions from the B cache to the A cache require a swap to be performed: the block from the B cache is moved into the A cache and

	Single	Assist		Victim		NTS		PCS	
Cache	A	A	B	A	B	A	B	A	B
Size	8/16K	8K	1K	8K	1K	8K	1K	8K	1K
Associativity	1/1	1	full	1	full	1	full	1	full
Replacement Policy	–/–	–	FIFO	–	LRU	–	LRU	–	LRU
move time	–	1		2		–		–	
latency to next level	18	–	18	18	–	18	18	18	18

Table 2.3: Characteristics of the five configurations studied. Times/ latencies are in cycles.

the block it evicts from the A cache is placed in the B cache. Normally, this operation cannot complete in a single cycle, as there is only a single, albeit dedicated, bus between the caches, and two elements need to be moved using the common bus. Thus, we can assign a latency of two cycles for a move between the caches for the Victim configuration or assign a one cycle latency and assume a 2 block wide bus; we assigned latency 2 in our experiments in this chapter, but vary the swap latency in Chapter IV. If there is an access to a block that is moving between caches, the trailing-edge effect seen by this latter access is properly accounted for by the LE cache model, i.e. an access to a moving block completes in less time than the swap time + cache hit time but incurs a latency greater than that of a cache hit.

Each of the multilateral configurations is discussed in more detail in Section 2.6. From these brief examples, however, it is easy to see that this modular, library-based approach to defining a cache configuration allows a significant range of configurations to be examined early in the design cycle.

2.6 Cache configurations

In each of the following multilateral cache configurations, both the A and B caches are checked in parallel on each memory access.

2.6.1 Assist cache

The Assist cache [22][23] (Figure 2.2i) is a multilateral design where the B cache is used as a “staging area” for data entering the A (main) cache. On a hit, the data is returned to the processor the next cycle, but remains in the cache in which it is found. On a miss, the block entering the L1 cache structure is placed into the B cache regardless of its reuse characteristics. Blocks in the B cache are managed in a FIFO fashion, where each block evicted from the B cache is placed in the A cache. A block evicted from the A cache due to a promotion returns to the next level of memory. By allowing blocks to enter L1 via a highly-associative buffer (the B cache), the Assist scheme allows multiple blocks that would map to the same set in A to reside in L1 concurrently for a short period of time, thereby reducing the amount of thrashing in that cache set. The Assist cache is particularly effective in eliminating ping-pong miss patterns, wherein a few blocks (more than the A cache associativity) in a set are referenced intensively in an interleaved fashion for a while and are then not referenced for a long time. However, its performance is degraded when an inactive block, which has spent its useful lifetime in B, is finally promoted and replaces and active block in A.

In the Assist implementation in the HP PA-7200 [22], a compiler hint is used to aid in keeping spatial-only data from polluting the A cache. Data blocks are tagged spatial-only if they are predicted to be use-once or the working set to which they belong is too large to be effectively cached (e.g. for streaming data). Blocks that have the spatial-locality hint bit set are returned to the next level of memory upon eviction from the B cache. The compiler has the burden of deciding whether data is spatial-only or not, and it is not clear whether a compiler can do it well. However, we do not model these compiler hints in this dissertation. The Assist cache as used here with no hints generally outperforms a modified Assist cache using hardware-tagging of spatial-only blocks [3], because the hardware-tagging mechanism sometimes mistakenly tags blocks that would become temporal if promoted as spatial-only, thereby preventing their promotion to the A cache. If effective means were available for providing compiler hints, they could benefit all of the multilateral schemes, as mentioned in Section 2.9.2.

As shown in Figure 2.3, the Assist cache can be implemented using the library routines provided.

2.6.2 Victim cache

The Victim cache (Figure 2.2ii), based on the scheme proposed in [18], is a multilateral design where the B cache is managed in a fashion akin to the victim buffer. On a memory access, desired data found in the A cache is returned to the processor the next cycle. If the desired data is found in the B cache, the desired block must first be “swapped” into the A cache — the block from the B cache is placed in the A cache and the block it evicts from A is placed in the B cache. Depending upon the amount of hardware dedicated to handling these swaps, a hit to the B cache may require varying amounts of time. In this study, we assume an additional 2 cycle latency for B cache hits, though we vary the swap latency in some later experiments which are presented in Chapter IV.

On a miss, the block entering the L1 cache structure from the next level of memory is placed in the A cache. The block evicted from A as the result of a new block’s arrival is placed in the B cache; blocks evicted from the B cache return to the next level of memory. Like the Assist cache, the Victim cache manages the cache state passively — recently evicted blocks are always saved in the auxiliary cache for faster access in case they are referenced in the near future; no active placement decision is made based on reuse information.

The Victim cache can also be implemented using the library routines provided. On a cache miss, a DO_SAVE_EVICTED is performed, so the new block is placed in the A cache and the evicted block is placed in the B cache (after the specified access latencies). On a B cache hit, a DO_SWAP is done to place the referenced data in the A cache, and to move the block it evicts from the A cache into the B cache.

2.6.3 NTS cache

The NTS cache, using the model in [6], which was adapted from the scheme proposed in [3], actively allocates data within L1 based on each block's usage characteristics. In particular, blocks known to have exhibited only nontemporal reuse are placed in B, while the others (presumably temporal blocks) are sent to A. This is done in the hope of allowing temporal data to remain in the larger A cache for longer periods of time, while shorter lifetime nontemporal data can for a short while be quickly accessed from the small, but more associative B cache.

The cache directory entry for each block in cache A and B is augmented with a temporality detector, consisting of one bit per word in the block, plus an additional temporality indicator bit. When the block is brought into cache, all the temporality detector bits are set to 0. On a block access, the bit in the temporality detector corresponding to the accessed word is set to 1. If that bit was previously set to 1, the temporality indicator bit for the block is set to 1, indicating temporal reuse of the block.

On a memory access, if the desired data is found in either A or B, the data is returned to the processor with 0 added latency, and the block remains in the cache in which it is found. On a miss, the block entering L1 is checked to see if it has an entry in the Detection Unit (DU). The DU contains temporality information about blocks recently evicted from L1 and is managed as follows. Each entry of the DU describes one block and contains a block address (for matching) and a T/NT bit (to indicate the temporality of its most recent tour). On eviction, a block is checked to see if it exhibited temporal reuse (i.e. if its temporality indicator bit is set to 1, indicating that some word in the block was referenced at least twice during this just-completed tour in the L1 cache structure), and its T/NT bit is set accordingly in the DU. If no corresponding DU entry is found for the evicted block, a new DU entry is created and made MRU in the DU structure. On a miss, if the new (miss) block address matches an entry in the DU, the T/NT bit of that entry is checked and the block is placed in A if it indicates temporal, and B if not. The DU entry is then made MRU in the DU so that it has a better chance of remaining in the DU for future allocation predictions. Thus, each creation or access of an entry in the DU is treated as a "use" and the DU is maintained with LRU replacement. If no matching DU entry is found, the miss block is assumed to be temporal and placed in A.

mlcache does not provide routines for managing a DU. The user can add routines that manage the DU and perform placement decisions based on the results of those routines. We added routines that maintain the DU as a separate, finite-sized fully-associative cache indexed by an address. Each entry in the DU contains an effective address and a T/NT bit which is set to 1 if the block at that address exhibited temporal reuse during its most recent cache tour, and 0 if not. This T/NT bit is used to predict the temporality of the next tour of this block. These routines are called within the `config.c` file when a cache miss is handled and used to decide whether to place the data in the A or B cache. Since this is a user-added routine, we can specify separate parameters for this module. For our experiments, we use a DU size of 32 entries.

2.6.4 PCS cache

The PCS cache [6] decides on data placement based on the program counter value of the memory instruction causing the current miss, rather than on the effective address of the block as in the NTS cache. Thus, the performance of blocks missed by individual memory accessing instructions, rather than individual data blocks, determines the placement of data in the PCS scheme.

The modeled PCS cache structure is chosen to be as similar as possible to the NTS cache (Figure 2.2iii). The cache directory entry for each block in cache A and B is augmented with the temporality detector bits, as in NTS, along with the program counter value of the instruction that brought the block to cache. The DU is accessed by finding an entry that matches the memory accessing instruction's program counter, but is updated in a manner similar to the NTS scheme. When a block is replaced, its temporality bit is copied into the T/NT bit of the entry corresponding to its PC value; if no such entry is found, one is created. When an instruction misses, the loaded block will be placed in the B cache if the instruction's PC value hits in the DU and the prediction (T/NT) bit indicates NT; otherwise the block is placed in the A cache. If the instruction misses in the DU the data is placed in the A cache.

Given the similarity of the NTS and PCS cache structures, we implemented the PCS cache using the routines that we created for the NTS cache, but simply indexed the

DU by the PC instead of by the effective address of its data. As with the NTS cache, we use a 32-entry DU for our experiments.

2.7 Implementation and testing of *mlcache*

We implemented five different cache configurations using the *mlcache* simulator: a direct-mapped single structure cache, an Assist cache, a Victim cache, an NTS cache, and a PCS cache. The latencies used for the timing simulation of these caches are shown in Table 2.3.

2.8 Simulation environment

A timing simulation of caches is of limited use without considering the latency-masking effects of processor execution. Thus, we integrated *mlcache* into the *SimpleScalar* [21] *sim-outorder* simulator by replacing *SimpleScalar's* data cache module with *mlcache*. Note that *mlcache* can be combined with any currently available instruction-level simulator, including *Talisman* [29], *SimICS* [30], *RCM_brisc* [20], and others. This flexibility is possible because *mlcache* maintains the state of the caches itself and does not take into account virtual memory or TLB effects. It models up to two cache structures and assumes a perfect memory thereafter, regardless of the number of level of caches beyond that. We chose the *sim-outorder* processor model because it performs out-of-order issue, execution, and completion on a derivative of the MIPS instruction set architecture, and is a well regarded processor simulator in general.

Since our study focuses on the effectiveness of a given data cache configuration in reducing memory access time, we designed a near perfect instruction supply mechanism

Fetch Mechanism	fetches up to 16 instructions in program order per cycle
Instruction Cache	perfect cache, 1 cycle latency
Branch Predictor	perfect branch prediction
Issue Mechanism	out-of-order issue of up to 16 operations per cycle, 256 entry re-order buffer (RUU), 128 entry load/store queue (LSQ); loads may execute when all prior store addresses are known
Functional Units	16 integer ALUs, 16 FP ALUs, 8 integer MULT/DIV, 8 FP MULT/DIV, 8 L/S units
Functional Unit Latency (total/issue)	integer ALU:1/1, integer MULT:3/1, integer DIV:12/12, FP adder:2/1, FP MULT:4/1, FP DIV:12/12. load/store:1/1
Data Cache	write-back, write-allocate, 32B lines, 8 read/write ports, non-blocking

Table 2.4: Processor and memory subsystem characteristics.

for our processor simulator. In addition, we provided ample resources for the instruction processing phase in order to maximize the effect of data cache performance. Table 2.4 details our chosen parameters and architectural assumptions. Since 16-way superscalar processors are likely in the near future, this processor configuration will shed light on the benefits of using multilateral caches in processor designs of coming generations.

2.8.1 Benchmarks

Table 2.5 shows the ten programs (5 integer and 5 floating point) selected from the SPEC95 benchmark suite [31] for this study. Each program, using the training data sets, was run to completion or through the first 1.5 billion instructions.

2.8.2 Experiments

The *mlcache* tool is capable of evaluating a large expanse of cache designs based upon different cache parameters; a listing of the available parameters is shown in Table 2.1. In order to present a simple cache configuration evaluation using *mlcache*, we

Program	Instruction Count (millions)	Memory References (millions)		Perfect Cache Performance	
		Loads	Stores	Cycle Count (millions)	IPC
SPEC95 Integer Benchmarks					
Compress	35.68	7.37	5.99	5.35	6.6644
Gcc	263.85	61.15	36.24	43.50	6.0648
Go	548.13	115.79	41.40	91.33	6.0049
Li	956.49	286.38	168.79	151.32	6.3210
Perl	1,500.00	396.82	269.83	232.89	6.4408
SPEC95 Floating Point Benchmarks					
Hydro2d	974.50	196.11	60.90	127.63	7.6353
Mgrid	1,500.00	528.07	21.53	194.84	7.6985
Su2cor	1,054.09	262.20	84.74	152.34	6.9192
Swim	849.92	205.18	58.44	113.02	7.5201
Wave5	1,500.00	321.87	133.26	318.69	4.7067

Table 2.5: The ten benchmarks and their memory characteristics.

chose to keep some of these parameters constant. The multilateral caches are all of the same size: the A cache is 8K and the B cache is 1K. While these caches may be small by today's standards, they are convenient for illustrative purposes. Previously evaluated multilateral cache designs were small, and their performance was found to rival the performance of direct-mapped caches of nearly twice the size. The choice of these cache sizes was driven by our use of the SPEC benchmarks — larger caches can hold the entire working set of these benchmarks, reducing the benefit of multilateral designs [10]. Given our choice of the widely used and well-understood SPEC benchmarks for these evaluations, we feel that these smaller cache sizes yield results that are indicative of the relative merits of the schemes on larger codes with larger caches.

The management of the data within the caches is dictated by the four multilateral designs we chose to evaluate (the management of data in the direct-mapped single structure cache is obvious). We chose to keep the latencies to memory constant over all configurations, as we are simply evaluating the effect of using each cache design in the same processor. The latencies of moving data between the A and B caches depends upon the cache design, as listed in Table 2.3.

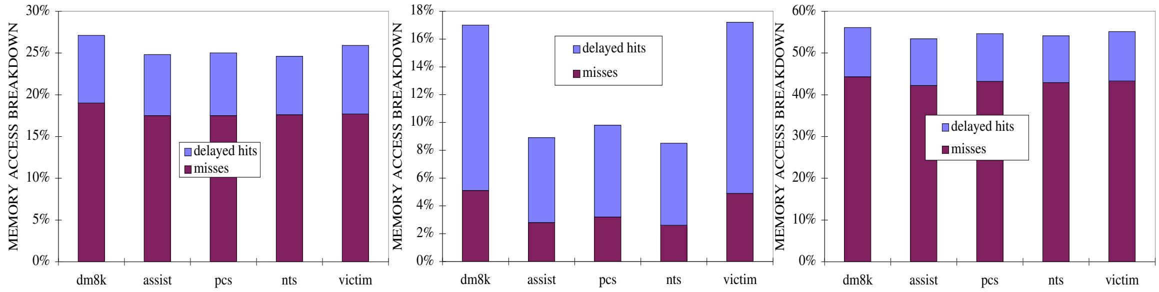
To get a feel for each configuration's performance, we tested each using a direct-mapped A cache and a fully-associative B cache; this configuration, like the Victim swap latency, is also varied in Chapter IV. This is also the configuration used in previous evaluations of the various cache designs, so this experiment helps gauge the correctness of the performance reported by *mlcache*. The following section discusses the results of the *mlcache* simulations.

2.9 Experimental results

2.9.1 Distribution of hits, misses, and delayed hits

Miss ratio is a first-order performance metric typically used to describe cache performance. However, incorporation of latencies introduces three categories of memory accesses: 1) (undelayed) hits to the cache, 2) misses to the cache, and 3) delayed hits. Hits and misses to the cache are defined simply as accesses to data that is resident/not resident in the cache at the time of access. Delayed hits are typically categorized as hits in behavioral cache simulators, as they do not cause any additional traffic between cache and memory. However, delayed hits typically experience a latency that is greater than the nominal hit latency due to latency-adding effects.

mlcache makes a distinction between these three access categories and reports the relative number of accesses in each category. Figure 2.4 shows the miss and delayed hit



i) compress **ii) go** **iii) hydro2d**

Figure 2.4: Memory access breakdown for the different cache configurations for three benchmarks: i) the compress benchmark, where delayed hits contribute moderately to the number of accesses that miss in the cache; ii) go, where delayed hits are prominent; and iii) hydro2d, where misses are dominant.

ratios for three benchmarks that show the impact of delayed hits: i) **compress**, ii) **go**, and iii) **hydro2d**. We see that the breakdown of references that do not hit in the cache can vary greatly; delayed hits are most prominent in **go**, misses in **hydro2d**, and **compress** is in between these two. Since misses generally have significantly larger latencies than delayed hits, two configurations with similar hit ratios may result in drastically different overall run times due to the breakdown of misses and delayed hits. For these experiments, the average delayed hit latencies for reads and writes were 9.89 and 11.17 cycles, respectively, compared to the nominal miss latency of 18 cycles and hit latency of 0 cycles[3]. However, in aggressive out-of-order pipelines, in contrast to miss latency, more of the latency of delayed hits may be masked by performing other useful work.

2.9.2 Relative Cache Effect Ratio (RCR)

An important metric for evaluating any cache management scheme is the cache hit/miss ratio. However, in OOO processors with multi-ported non-blocking caches, effective memory latencies (as seen by the processor) vary according to the number of outstanding

3. Recall that 1 cycle for memory access is already built into the processor pipeline model. These latencies refer to additional cycles.

miss requests. Since the main focus of this study is to evaluate the effectiveness of the L1 cache structure using special management techniques, the Relative Cache Effect Ratio (RCR) was developed [7]. The RCR for a given processor running cache configuration *X* relative to cache configuration *base*, is given by:

$$RCR_X = \frac{Cycle\ Count_X - Cycle\ Count_{Perfect\ Cache}}{Cycle\ Count_{base} - Cycle\ Count_{Perfect\ Cache}}$$

where Cycle $\text{Count}_{Perfect\ Cache}$ is the total number of cycles needed to execute the same program on the same processor with a perfect cache configuration. RCR, a normalized metric between the base cache and the perfect cache, is 1 for the base cache configuration and 0 for the perfect cache configuration. Cache configurations that perform better than the base have RCR between 0 and 1, with lower RCR being better. A cache configuration that performs worse than the base has RCR > 1. RCR gives an indication of the finite cache penalty reduction obtained when using a given cache configuration. RCR mirrors the performance indicated by speedup numbers, but isolates the cache penalty cycles from total run time and rescales them as a fraction of the penalty of a traditional (base) cache. It thus gives a direct indication of how well the memory subsystem performs relative to an ideal (perfect) cache. Table 2.5 shows the number of cycles required to execute the code on our SimpleScalar processor configuration with a perfect cache (i.e. all memory accesses are satisfied at the end of the cycle in which they commence).

The RCR for each of the cache configurations, using the direct-mapped 8K single structure cache as the base, is shown in Figure 2.5. While performance varies among the various configurations, several trends emerge. First, multilateral designs can greatly improve cache performance by reducing the overall number of cycles required to access memory. For **go** and **perl**, multilateral designs can cut the total finite cache penalty to less

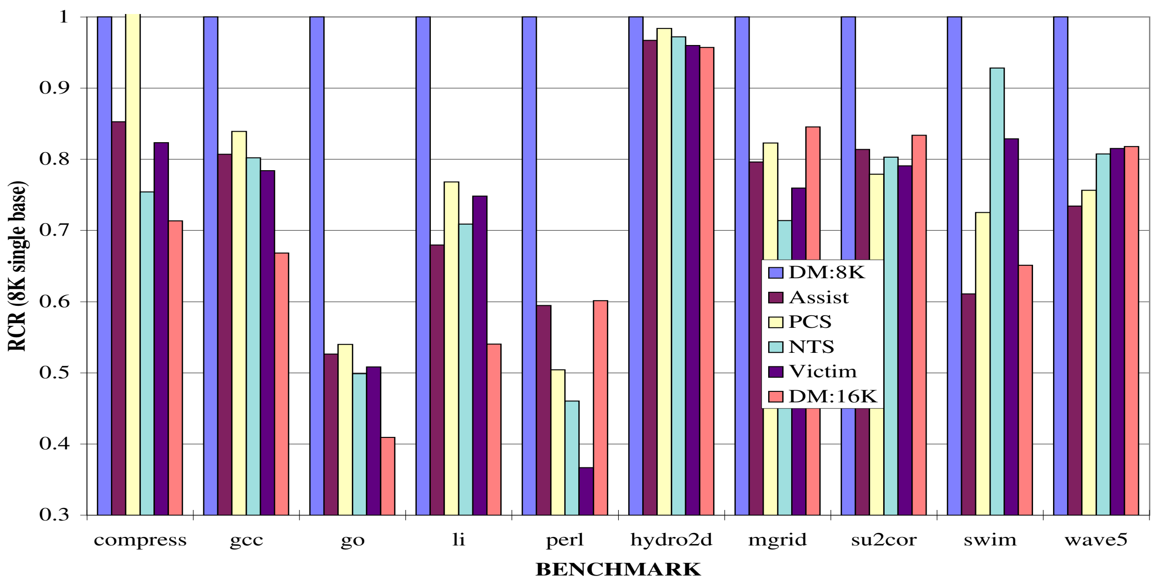


Figure 2.5: Relative Cache Effect Ratio comparison for the two direct-mapped and four multilateral configurations.

than half of the base 8K direct-mapped cache. Further, the multilateral designs often approach, or even exceed, the performance of a larger, 16K direct-mapped cache — the 16K cache performs best only in **compress**, **gcc**, **go**, **li**, and **hydro2d**, where the added capacity of the larger cache is more beneficial than the improved cache management provided by the multilateral schemes.

The Assist cache used in the HP PA-7200 made use of a compiler-supplied hint. Though the hint was not accounted for in this study, the Assist, NTS, and PCS caches could all benefit from compiler hints regarding proper management for particular cache blocks. For instance, the NTS and PCS caches could use compiler hints to place data for which no DU entry exists into the B cache, instead of into the A cache by default — if the compiler knew that the accessed data was nontemporal, this hint could reduce pollution of the A cache with nontemporal data. Thus, it is likely that with compiler assistance, the performance of the Assist, NTS, and PCS caches would each improve over what these experiments have shown.

2.9.3 Block tour information

In general, reducing the number of tours through the cache for a given benchmark results in improved overall performance — fewer tours indicate that data is used more often during each tour, resulting in fewer fetches to the next level of memory. *mlcache* provides statistics for block tours to each cache. For configurations that do not transfer data directly between the A and B cache (i.e. single, NTS, and PCS caches) the number of block tours equals the number of misses. However, for configurations that allow data movement between the caches like the Assist and Victim caches, the total number of tours also counts the number of times that a block is moved between the A and B caches. Thus, for all configurations, total tours reports A cache tours plus B cache tours.

To more accurately explain the performance of the Assist and Victim caches with regard to block tours, we must also account for the number of tours caused by data movement between the caches. These tours are caused by efforts to improve data reuse and are not as costly as accesses to the next level of memory. We can subtract the number of tours due to a swap of two blocks between the caches and migrations (saves) of a block from one cache to the other from the total tours to obtain L1 tours (i.e. the number of fetches from the next level of memory). Thus, in general, L1 tours correspond to misses (trips to L1 from L2), whereas total tours correspond to misses plus 2x the number of swaps (since a swap causes two tours to commence) plus the number of saves.

Table 2.6 shows the tour information for two benchmarks with differing performance using multilateral caches, **go** and **hydro2d**, on three cache configurations, an 8K single structure cache, an NTS cache, and a Victim cache. For **go**, the multilateral designs result in large performance improvements over the base 8K cache, with RCRs of

	RCR	Total Tours	Swaps	Saves	L1 tours	Total % References to each Tour Group			
						NTNS	NTS	TNS	TS
8K:1w	1.00	16.1M	–	–	16.1M	3.91	1.95	24.0	70.1
NTS	0.49	8.1M	–	–	8.1M	1.41	1.05	17.8	79.8
Victim	0.51	33.7M	9.9M	7.0M	7.0M	9.14	1.84	22.7	66.3

i) go

	RCR	Total Tours	Swaps	Saves	L1 tours	Total % of References to each Tour Group			
						NTNS	NTS	TNS	TS
8K:1w	1.00	28.7M	–	–	28.7M	0.13	29.4	6.8	63.7
NTS	0.97	27.4M	–	–	27.4M	0.03	28.8	4.9	66.2
Victim	0.96	57.6M	1.4M	27.4M	27.5M	10.2	26.3	6.1	57.3

ii) hydro2d

Table 2.6: Block tour information for three cache configurations running go (top) and hydro2d (bottom).

approximately half the cache effect of the base cache, i.e. each multilateral design requires only about half the tours incurred by the base cache through the L1 cache. The number of L1 tours is calculated by subtracting the number of saves and 2x the number of swaps performed from the total number of block tours reported by *mlcache* — for the direct-mapped and NTS caches, since they incur no data movement between the caches, the number of L1 tours equals the total number of tours

Note that though the Victim cache incurs the fewest L1 tours of the three configurations running **go**, its RCR is higher than that of NTS. This is due to the added two cycle latency for each swap — though the Victim cache has 1.1M fewer L1 tours than the NTS cache, each of the 9.9M swaps adds two cycles to the access latency, and potentially to the execution time. If the latency of the swap is reduced or eliminated, the Victim cache's performance will improve (as shown in Section 4.4.1).

For **hydro2d**, the performance of the three configurations is much closer, with the RCRs differing by less than 5%. This performance similarity is also reflected in the number of L1 tours experienced by each configuration — approximately 28M each. The resulting RCRs of the multilateral designs are close; even though the NTS cache has slightly fewer L1 tours and no swaps or saves, the RCR of the Victim cache is slightly lower.

2.9.4 Block reuse information

mlcache also provides reuse information for each block tour. We divide block usage into four categories, as discussed in Section 2.2: 1) nontemporal nonspatial (NTNS), 2) nontemporal spatial (NTS), 3) temporal nonspatial (TNS), and 4) temporal spatial (TS). Good cache configurations should generally result in fewer tours and a higher percentage of data references to blocks making TS tours. NTNS and NTS tours are problematic; they may cause excessive cache pollution, and should be minimized if possible.

We see in Table 2.6 that the NTS cache does a particularly good job of managing the cache state, resulting in a higher percentage of TS tours in **go**. Furthermore, the NTS cache reduces the percentage of references to NTNS and NTS data relative to the base cache, which also contributes to its performance. The Victim cache's reuse information is less indicative of its performance. Blocks in a Victim cache may experience many tours between A and B while still remaining within the L1 structure. Thus, though Victim has a lower percentage of TS tours and a higher percentage of NTNS tours compared to the base cache, its performance is still better than that of the base cache. However, the Victim cache manages its state passively, so block reuse information is more of a statistic than an aid to cache management, as in the NTS scheme.

The lack of performance improvement of the multilateral designs running **hydro2d** can also be explained by the high percentage of nontemporal (NTNS + NTS) accesses that each configuration experiences. Data in these categories often pollute the cache, as they are not reused with high frequency and often evict more useful, temporal data. This information can be used to design cache configurations and management schemes than can help attain higher performance when running benchmarks like **hydro2d**.

Thus, block tour and reuse information can be used to evaluate and explain the performance of target caches running target applications. Furthermore, the statistics provided by *mlcache* can help derive new cache designs that can better handle the reference patterns exhibited by the target workloads.

2.10 Summary

mlcache is a flexible, multilateral cache simulator developed to help designers make cache configuration decisions early in the design cycle that would best aid in attaining the desired performance goals of the target processor. *mlcache* is an event-driven, timing-sensitive simulator based on the Latency Effects cache timing model. It can be easily configured to model various multilateral cache configurations by using its library of cache state and data movement routines. The simulator can be easily joined to a wide range of event-driven processor simulators such as *RCM_brisc*, *Talisman*, *SimICS*, and *SimpleScalar*. For this study, we integrated *mlcache* into the *SimpleScalar sim-outorder* processor simulator.

We showed implementations of five different cache configurations and their resulting performance when running ten of the SPEC95 benchmarks. These configurations included a direct-mapped single structure cache and four multilateral caches: an Assist cache, a Victim cache, an NTS cache, and a PCS cache. Each was easily modeled in *mlcache* using the library routines provided.

mlcache provides many statistics which can help explain the performance of proposed cache configurations when running target workloads. Information regarding hits, misses, and delayed hits tells of the program's memory access characteristics, while block tour and reuse information tells of the actual data usage within each program. These statistics can be used to explain the performance of each cache configuration as well as to help derive future cache designs that better handle the reference streams presented by the target workloads.

CHAPTER III

ACTIVE MANAGEMENT OF DATA CACHES BY EXPLOITING REUSE INFORMATION

3.1 Introduction

In this chapter we evaluate the performance of three cache schemes that perform active block allocation and compare their performance to one another and to traditional single structure caches. We implement the MAT [4], NTS [3], and PCS [6] schemes using hardware that is as similar as possible in order to do a fair comparison of the block allocation algorithms that each uses. We find that making placement decisions based on effective address-based block reuse, as in the NTS scheme, outperforms the macroblock-based and PC-based approaches of MAT and PCS, respectively. Nevertheless, all three schemes perform comparably to larger direct-mapped caches and better than associative caches of similar size.

We then examine the performance of optimal and near-optimal multilateral caches to determine the performance potential of multilateral schemes. Optimal and near-optimal schemes excel in block replacement decisions. Since they can swap blocks freely between A and B, their block allocation decisions are not critical and simply follow as a direct consequence of the replacement decisions. We compare the performance of two implemented multilateral schemes to the near-optimal scheme in order to assess the reasons for their particular level of performance. This analysis shows that the implemented schemes might be made to perform better by improving their block allocation choices and incorporating more robust replacement mechanisms.

3.2 Active cache management

Active cache management can be used to improve the performance of a given size cache structure by controlling the data placement and management in the cache to keep the active working set resident. Active management of caches consists of two parts: allocation of blocks within the cache structure on a demand miss[4] and replacement of blocks currently resident in the cache structure. Block allocation in today's caches is passive and straightforward: blocks that are demand fetched are placed into a set within the cache structure according only to their address rather than their behavior[5]. Thus, this decision does not take into consideration the block's usefulness or usage characteristics. Examining the past history of a given block is one method of aiding in the future allocations of the block. Decisions can range from simply not caching the miss block (bypassing) to placing the block in a particular portion of the cache structure, with the hope of making best use of the miss block and the blocks that remain in the cache.

Simple block replacement policies are used to choose a block for eviction in today's caches, and this choice is often suboptimal. For a multilateral cache, the allocation and replacement problems are coupled. In particular, the blocks that are available for replacement are a direct consequence of the allocation of a demand-missed block.

---

4. Allocation decisions for blocks not loaded on a demand miss, e.g. block prefetching and cache bypassing schemes, are not considered here. However, schemes that make proper bypass decisions and allocation decisions for prefetched data can further improve upon the performance of the schemes evaluated herein.
5. We consider only write-allocate caches in this dissertation. Write no-allocate caches follow a subset of these rules, where writes are not subject to these allocation decisions.

Recently, several approaches to more efficient management of the L1 data cache via block allocation decisions have emerged in the literature: NTS [3], MAT [4], Dual/Selective [5], and PCS [6]. However, none of these approaches makes sophisticated block replacement decisions, and instead relegates these decisions to their respective cache substructures.

3.2.1 The NTS model

The NTS (nontemporal streaming) cache [3] is a location-sensitive cache management scheme that uses hardware to dynamically partition cache blocks into two groups, temporal (T) and nontemporal (NT), based on their reuse behavior during a past tour. A block is considered NT if during a tour in L1, no word in that block is reused. Blocks classified as NT are subsequently allocated in a separate small cache placed in parallel with the main L1 cache; all other blocks (those marked T and those for which no prior information is available) are handled in the "main" cache. Data placement is decided by using reuse information that is associated with the effective address of the requested block. The effectiveness of NTS in reducing the miss ratio, memory traffic, and the average access penalty has been demonstrated primarily with mostly numeric programs.

3.2.2 The MAT model

The MAT (memory address table) cache [4] is another scheme based on the use of effective addresses. However, as opposed to the T/NT tagging of individual cache blocks as in NTS, MAT dynamically partitions regions of memory into two groups based on their frequency of reuse. Memory regions become tagged as either Frequently or Infrequently Accessed, and a memory address table is used to keep track of this reuse information. The granularity for grouping is a *macroblock,* defined as a fixed size partition of the memory space. All blocks within a particular macroblock are treated as if they have the same usage pattern characteristics. Blocks in macroblocks that are determined to be Infrequently Accessed are allocated in a separate small cache. This scheme has shown significant speedups over generic caches due to improved miss ratios, reduced bus traffic, and a resulting reduction in the average data access latency. The performance of MAT is best for programs in which the reference behavior of a given datum is well-correlated with the macroblock to which it maps. A detailed description of MAT's implementation is presented in Section 3.3.

3.2.3 The Dual cache/Selective cache model

The Dual Cache [5] has two independent cache structures, a spatial cache and a temporal cache, where the temporal cache has a smaller line size than the spatial cache. A locality prediction table is used to predict the locality of the referenced block and place it in the corresponding cache structure. Each entry in the prediction table contains information about a recently executed load/store instruction, which is used to predict the locality of blocks that are subsequently accessed and missed by those load/store instructions. A miss block that is predicted to be neither spatial nor temporal does not find a place in the cache and bypasses the cache. This method is more useful in handling vector operations which have random access patterns or very large strides and introduce self-interference. Unlike MAT, NTS, and PCS (discussed below), its two caches do not necessarily maintain disjoint contents. If the required data is found in both caches it is read from the temporal cache or written into both in parallel. In order to overcome this replication and coherence problem, the authors proposed a simplified version of the Dual

Cache, called the Selective Cache. The Selective Cache has only one data store like a conventional cache, but incurs more hardware cost due to its locality prediction table, as in the Dual Cache. Only data exhibiting spatial locality or temporal locality that is not self-interfering is cached. For most of the benchmarks in their study, this scheme was shown to perform better than a conventional cache of the same size. The Selective Cache itself is an improvement of the Bypass Cache [33], which relies on compiler hints to decide whether a block is to be cached or bypassed.

3.2.4 The PCS model

The PCS (program counter selective) cache [6] is a multilateral cache design that evolved from the CNA cache scheme [2]. The PCS cache decides on the data placement of a block based on the program counter value of the memory instruction causing the current miss, rather than on the effective address of the block as in the NTS cache. Thus, in PCS, the tour performance of blocks recently brought to cache by this memory accessing instruction, rather than the recent tour performance of the current block being brought to the cache, is used to determine the placement of this block. The performance of PCS is best for programs in which the reference behavior of a given datum is well-correlated with the memory-referencing instruction that brings the block to cache.

3.2.5 Other multilateral cache schemes

Several other cache schemes can be considered multilateral caches, such as the Assist cache [22][23] used in the HP PA-7200, and the Victim cache [18]. However, neither of these schemes actively manage their cache structures using reuse information obtained dynamically during program execution. The Assist cache uses a small data store as a staging area for data entering the L1 and potentially prevents data from entering the L1 when indicated by a compiler hint. The Victim cache excels in performance when a majority of the cache misses are conflict misses which result from the limited associativity of the main cache; the buffer in the Victim scheme serves to dynamically increase the associativity of a few hot spots in the (typically direct-mapped) main cache. While both schemes have been shown to perform well [32], they each require a costly data path between the two data stores to perform the data migrations they require. These schemes both inherently rely on the inter-cache data path, as they use no previous tour information for actively deciding which data store to allocate a block to. While the Victim cache has been shown to perform well relative to the above actively-managed schemes when the main cache is direct-mapped [6], in this chapter we evaluate only multilateral schemes that use dynamic information to allocate data among two data stores with no direct data path between them.

3.3 Simulation methodology and cache models used

We use the same simulation environment and benchmarks as in Section 2.8. However, we configure the cache structures to be larger for this study in order to evaluate the effect of increased cache size on the performance of multilateral schemes (in particular, NTS and PCS). We evaluate the performance of three of the multilateral schemes, MAT, NTS, and PCS, with 16K direct-mapped A caches and 2K fully-associative B caches and compare their performance to three traditional, single structure caches: a 16K direct-mapped cache, a 16K 2-way associative cache, and a 32K direct-mapped cache. The configurations of the evaluated caches are shown in Table 3.1.

	Single	MAT		NTS		PCS	
Cache	A	A	B	A	B	A	B
Size	16K/16K/32K	16K	2K	16K	2K	16K	2K
Associativity	1/2/1	1	full	1	full	1	full
Replacement Policy	–/LRU/–	–	LRU	–	LRU	–	LRU
Latency to next level	18	18	18	18	18	18	18

Table 3.1: Characteristics of the four configurations studied. Times/latencies are in cycles.

Performing a realistic comparison among the program counter and effective address schemes requires detailed memory simulators for the MAT, NTS and PCS cache management schemes described above. We chose to omit the Selective Cache, as its block allocation decisions are similar to those made by PCS, while PCS' hardware implementation is simpler. We used the *mlcache* tool to place all the management schemes on the same platform within a uniform multilateral environment, thereby ensuring a fair comparison. Each of the configurations includes a 32-entry structure that stores reuse information, as described for each scheme.

The implementations of the NTS and PCS schemes were described in detail in Section 2.6. The MAT cache [4] structure has a Memory Address Table (MAT) for keeping track of reuse information and for guiding data block placement into the A or B cache of the L1 structure. In this implementation, the MAT is a 32-entry fully-associative structure (like the DU in NTS and PCS). Note, however, that the original implementation of MAT (reported in [4]) used a 1K entry direct-mapped table. Each MAT entry consists of a macroblock address and an n-bit saturating counter. An 8-bit counter and a 1K macroblock size is used here, as in the original study.

On a memory access, caches A and B are checked in parallel for the requested data. At the same time, the counter in the corresponding MAT entry for the accessed block is incremented; if there is no corresponding entry, one is created, its counter is set to 0, and the LRU entry in the MAT is replaced. This counter serves as an indicator of the "usefulness" of a given macroblock, and is used to decide whether a block in that macroblock should be placed in the A or B cache during its next tour.

On a cache miss, the macroblock address of the incoming block is used as an index into the MAT. If an entry exists, its counter value is incremented and compared against the decremented counter of the macroblock corresponding to the block that would be replaced if the incoming block were to be placed in the A cache. The counter of the resident data is decremented to ensure that it can eventually be replaced; this counter value will continue to decrease if the macroblock is not reaccessed often enough and it continues to conflict with more recently accessed blocks. If the counter value of the incoming block is higher than that of the conflicting block currently in cache A, the incoming block replaces this block in the A cache. This situation indicates that the incoming block is in a macroblock that has shown more "usefulness" in earlier tours than the macroblock in which the conflicting block resides, and the incoming block is thus given a higher priority for residing in the larger main cache. If the counter value of the incoming block is less than that of the currently resident conflicting block, the incoming block is placed in the smaller B cache.

Finally, if no entry corresponds to the incoming block, the block is placed in the A cache by default and a new entry is created for it in the MAT, with its counter initialized to zero. If no entry corresponds to the conflicting block currently in cache A, its counter value is assumed to be 0, permitting the new block to replace it easily. When no entry is found in the MAT for a resident block in cache A, another macroblock that maps to the same set in the MAT must have been accessed more recently, and the current block is therefore deemed less likely to be used in the near future.

As with the NTS and PCS schemes, there is no direct data path between the A and B caches. Unlike those schemes, however, the MAT structure is updated for every access to the cache instead of only on replacements.

The three schemes are configured to be as similar as possible to one another in structure so that their performance differences can be attributed primarily to differences among the block allocation decisions that they make.

3.4 Experimental results

Miss ratio is often used to rank the performance benefits of particular cache schemes. However, miss ratio is only weakly correlated with the performance of latency-masking processors with non-blocking caches. Furthermore, it fails to capture the latency-adding effect of delayed hits on overall performance. Two programs exhibiting similar miss ratios may have quite different overall execution times due to differing numbers of delayed hits and the extent of latency masking, as shown in [32].

To avoid oversimplifying a cache scheme's impact on overall performance, we instead concentrate on two metrics from timing-sensitive experiments: overall speedup relative to a base cache and the Relative Cache Effect Ratio (RCR) described in Section 2.9.2.

3.4.1 Miss ratio

Table 3.2 shows the miss ratios for each of the six cache configurations when running the ten benchmarks described in Section 2.8, and Figure 3.1 shows the

	MAT	NTS	PCS	16K:1w	16K:2w	32K:1w
Compress	0.220	0.219	0.221	0.239	0.217	0.215
Gcc	0.074	0.070	0.086	0.112	0.061	0.071
Go	0.031	0.021	0.024	0.070	0.038	0.035
Hydro2d	0.479	0.474	0.494	0.484	0.488	0.452
Li	0.041	0.043	0.050	0.062	0.038	0.029
Mgrid	0.107	0.101	0.107	0.107	0.096	0.098
Perl	0.060	0.032	0.033	0.062	0.044	0.051
Su2cor	0.256	0.257	0.257	0.269	0.248	0.244
Swim	0.220	0.230	0.167	0.228	0.224	0.161
Wave	0.326	0.281	0.249	0.348	0.297	0.271

Table 3.2: Miss ratios of the six cache schemes running the ten benchmarks.

corresponding speedup relative to a 16K direct-mapped cache. Naively, we might assume that when comparing two configurations for a particular application, a higher miss ratio would imply a lower speedup, and that (since cache stalls account for only a portion of the run time) the relative speedup would be less than the relative miss ratio. However, a comparison of Table 3.2 and Figure 3.1 shows that this assumption is not valid. In **compress**, for example, the miss ratio of PCS is about 1.01x that of NTS, but its run time is about 1.04x longer. In **gcc**, the 32K direct-mapped cache actually has a higher miss ratio, but less run time than the 16K 2-way associative cache. In **swim**, NTS has a higher miss ratio, but less run time than the MAT, 16K and 32K direct-mapped, and 16K 2-way caches. Thus, relative miss ratio alone is an inadequate indicator of relative performance; latency masking, miss latency overlap, and delayed hits must be incorporated in a timing model to get an accurate performance assessment. Therefore, we concentrate our performance analysis on latency-sensitive metrics, such as speedup and RCR.

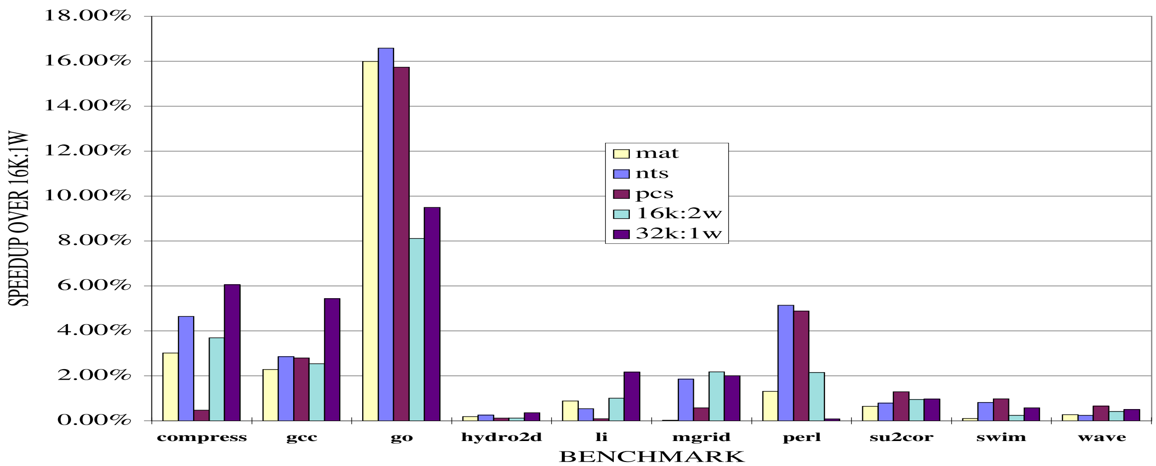


Figure 3.1: Overall execution time speedup for the five evaluated cache schemes, relative to a single direct-mapped, 16K cache (16k:1w).

3.4.2 Speedup

The speedup achieved by each scheme for each program is shown in Figure 3.1, where the single 16K direct-mapped (16k:1w) cache is taken as the base. Overall, the speedup obtained by using the multilateral cache schemes ranges from virtually none in **hydro2d** to just over 16% in **go** with NTS. Clearly, some of the benchmarks tested do not benefit from any of the improvements offered by the cache schemes evaluated, i.e. better management of the L1 data store by the multilateral schemes, increased associativity of a single cache (16k:2w), or a larger cache (32k:1w). In benchmarks where there is appreciable performance gain over the base cache, the multilateral schemes often perform as well as or better than either a higher-associative single cache or a larger direct-mapped cache. In **compress** and **gcc**, the benchmarks' larger working sets benefit from the larger overall cache space provided by the 32K direct-mapped structure, although even for these benchmarks the multilateral schemes are able to obtain a significant part of the performance boost via their better management of the cache. Despite their smaller size, the multilateral caches generally perform well compared to the larger 32K direct-mapped cache and are generally faster than the 2-way associative cache. In the multilateral schemes, the larger direct-mapped A cache offers fast access, and the smaller, more associative B cache can still be accessed quickly due to its small size. Our experiments show that using an 8-way associative B cache instead of a fully-associative B cache would reduce performance by less than 1%.

Among the multilateral schemes, we see that the NTS scheme provides the greatest speedup in all benchmarks except for **li** (where MAT performs best), **su2cor**, **swim**, and **wave** (where PCS performs best), but the best multilateral cache speedups are only on the order of 1% for these four benchmarks. Both the MAT and PCS schemes can perform well when the grouped blocks exhibit similar reuse behavior on consecutive tours through the cache. The NTS scheme may, however, fail to detect these reuse patterns because it correlates its reuse information to individual cache blocks, as opposed to macroblock memory regions in MAT or to memory accessing instructions in PCS. Thus, the MAT and PCS schemes can perform well if programs exhibit SIMD (single-instruction, multiple-data) behavior, where the reference behavior of nearby memory blocks or blocks referenced by the same memory accessing instruction may be a better indicator of reuse behavior than the usage of an individual block during its last tour. However, the NTS scheme is still competitive in these three benchmarks, and thus gives the best overall performance of these schemes over the full suite of benchmarks.

3.4.3 RCR performance

Figure 3.2 shows the RCR performance of MAT, NTS, PCS, and the two single structure caches, where the 16K direct-mapped cache serves as the base for comparison. We see here that NTS and PCS eliminate more than 50% of the finite cache penalty experienced by **go** and **perl**. In **compress**, **gcc**, and **li**, the 32K single structure direct-

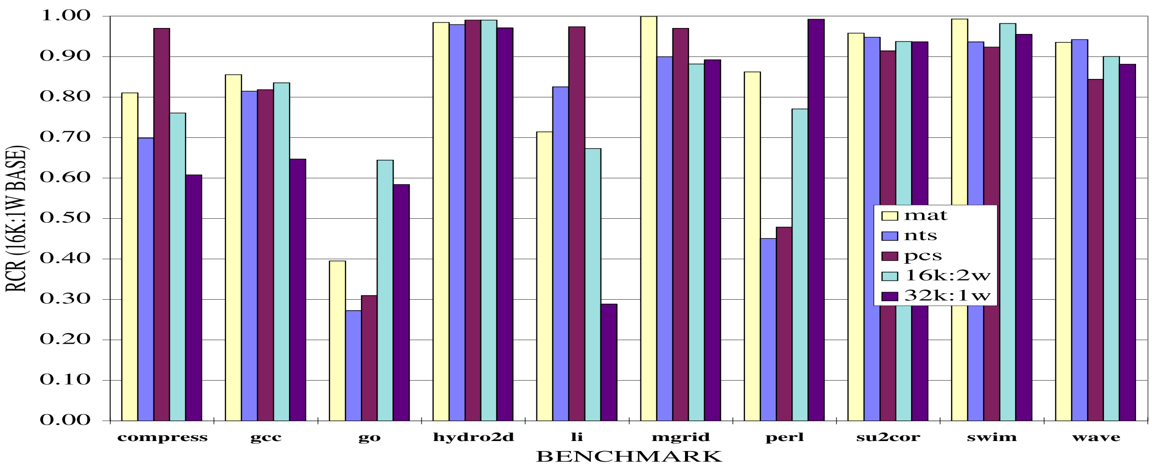


Figure 3.2: RCR performance of the evaluated configurations running the ten benchmarks. RCRs near 1.0 have performance similar to the base 16K direct-mapped cache while RCRs closer to 0 approach the performance of a perfect cache.

---

mapped cache performs best. However, the difference in RCR between it and the best-performing multilateral scheme is not very large, except for **li**, where it reduces the finite cache penalty more than twice as much as MAT, the best multilateral scheme for this benchmark. None of the caches show a significant improvement in RCR for the five remaining benchmarks (**hydro2d**, **mgrid**, **su2cor**, **swim,** and **wave**).

In some instances, one multilateral scheme can experience poor performance, e.g. MAT in the **perl** benchmark, relative to the other multilateral schemes. The block allocation scheme of MAT is not well matched to the characteristics of the **perl** benchmark. If many of the blocks are short-lived, but frequently accessed, those blocks will be placed in the smaller, 2K fully-associative cache. If this behavior continues, many blocks will contend for space in the smaller 2K fully-associative cache while the larger, 16K cache is badly underutilized. Such a phenomenon can occur in any of these multilateral cache schemes; in the extreme, the multilateral scheme's performance may degrade to that of the B cache by itself. This performance degradation can be addressed by improved block allocation mechanisms, as discussed in Section 3.5.

3.4.4 Performance differences and their causes

Though MAT and NTS both make block allocation decisions based on the effective address of the block being accessed, their performance differs. MAT may make poor allocation decisions when either the missed block or the block it would replace in cache A has a markedly different desirability than the perceived desirability of the macroblock in which it resides. When such a disparity in desirability occurs, a block-based desirability mechanism, such as that used in NTS, will perform better. Table 3.3 presents a tour analysis for two contrasting benchmarks, **go** and **su2cor**. The performance of NTS relative to MAT is well demonstrated in **go**. Not only does NTS appear to manage the tours in this program better, but it actually reduces the number of tours that MAT experiences by 32%. In the case of **su2cor**, where the performance difference between MAT and NTS is small (in terms of RCR), it is clear that the application itself has much more nontemporal spatial data (> 17%), and neither scheme reduces the tours seen by a 16K direct-mapped cache by more than 6%.

The tour analysis for the two single structure caches is also shown for these two widely disparate benchmarks. The analyses used to compare MAT and NTS can also be used to compare multilateral caches against single structure caches. In **go**, the 32K direct-mapped cache has the best single structure cache performance; though it has a similar percentage of TS tours as the NTS cache, it has substantially more overall tours, and thus worse overall performance. As with MAT and NTS in **su2cor**, the 2-way associative and 32K direct-mapped caches fail to significantly improve performance over the base cache due to the high percentage of NTS data accessed.

NTS performs better than PCS overall in both speedup and RCR. The different

Cache Management Scheme	Total # of Tours	% Reduction in Tours	Total Percentage of References to each Tour Group			
			NTNS	NTS	TNS	TS
16K:1w	6,694,690	-	1.27	1.00	11.43	86.29
MAT	2,807,282	58.06	0.48	0.63	5.63	93.26
NTS	1,894,991	71.69	0.21	.0.51	4.70	94.58
16K:2w	3,932,345	41.26	0.57	0.61	8.99	89.84
32K:1w	3,450,640	48.46	0.65	0.61	5.54	93.20

i) go

Cache Management Scheme	Total # of Tours	% Reduction in Tours	Total Percentage of References to each Tour Group			
			NTNS	NTS	TNS	TS
16K:1w	24,848,705	-	0.35	17.98	7.77	73.90
MAT	23,975,325	3.51	0.11	16.96	5.96	76.02
NTS	23,471,060	5.54	0.23	17.36	7.77	74.64
16K:2w	22,824,448	8.15	0.09	17.15	6.95	75.81
32K:1w	23,670,694	4.74	0.26	17.14	6.97	75.63

ii) su2cor

Table 3.3: Tour analysis for go (top) and su2cor (bottom).

basis for decision making used by PCS and NTS (PC and effective address, respectively), results in different performance. The PCS scheme may place a block suboptimally in the cache since its placement is influenced by other blocks previously referenced by the requesting PC. For example, a set of blocks may be brought into the cache by one instruction at the beginning of a large routine. These blocks may be reused in different ways during different parts of program execution (e.g. nontemporal during an initialization phase, but temporal during the main program's execution). All of these blocks' usage characteristics may be attributed to a single entry in the DU, tied to the PC of the instruction that brought the blocks to the cache. When each of these tours end, the instruction's entry in the DU is updated with that particular tour’s behavior, directly affecting the placement of the next block requested by this instruction. In effect, the allocation decisions of PCS are influenced by the most recently replaced block that is associated with the load instruction in question; if the characteristic of those most recently replaced blocks is not persistent, as discussed in Section 3.5.3.4, the allocation decisions made by PCS for this load instruction will vary often, potentially degrading performance.

However, program counter management schemes may be good if a given instruction loads data whose usage is *strongly biased* [34] in one direction, i.e. if these tours are almost all temporal or almost all nontemporal. In this case, accurate behavior predictions for future tours will result in good block placement for that instruction. However, if the data blocks loaded by the instruction have differing usage characteristics (i.e. *weakly biased* [34]) then placement decisions of its blocks will be poor.

Block usage history (T/NT) is kept in a single bit (NTS and PCS); macroblock frequency in an *n*-bit counter (MAT). Reducing the counter size in MAT generally leads to decreased performance [4]. However, a similar study for NTS and PCS (keeping tour history using a 2- or 3-bit counter) showed virtually no performance benefit over the 1-bit scheme.

3.5 Using reuse information in data cache management

Each of these multilateral schemes operates on the assumption that reuse information is useful in actively managing the cache. In this section, we assess the value of reuse information for making placement decisions in multilateral L1 cache structures. We first examine optimal cache structures to determine how they exploit reuse information in cache management. We then outline the experiments performed to validate the use of reuse information and compare the performance of multilateral schemes to the performance of near-optimally managed caches of the same size.

3.5.1 Optimally and near-optimally managed caches

Belady's MIN [35] is an optimal replacement algorithm for a single structure cache, i.e. it results in the fewest misses.[6] While it is interesting to see how reuse information is used to manage a single cache, we are interested in determining how an optimal replacement algorithm for a multilateral cache makes replacement decisions, and how reuse information might best be exploited. However, no direct extension of MIN to multilateral caches is known. The only exception is where both the A and B caches of a multilateral configuration are fully-associative and blocks are free to move between the two caches as necessary in order to retain the most useful blocks in the cache structure; this multilateral cache is, however, degenerate as it reduces to a single fully-associative cache of size equal to the total of cache A plus cache B. In this case, MIN can be used to optimally manage the hardware-partitioned fully-associative single cache.

We refer to Belady's MIN algorithm, when applied to the dual fully-associative caches, as *opt*. While *opt* gives an upper bound on the performance of a multilateral cache of a given size and associativity, comparing *opt* to the implementable schemes does not yield a direct comparison of replacement decisions based on reuse information. Since multilateral caches typically have caches A and B of differing associativity, the performance difference between the implementable schemes and *opt* may be due not only to replacement decisions, but also to mapping restrictions placed on the implementable

6. Note that we do not consider timing models in this section, for which MIN is not an optimal algorithm.

schemes by their limited associativity. We would instead like to compare the performance of the implementable schemes to an optimally managed multilateral configuration where the A and B caches are of differing associativity in order to better attribute the differences in placement and replacement decisions to the management policy itself, rather than to the associativity of the configuration.

Pseudo-opt [36] is a multilateral cache management scheme for configurations where the associativity of A is no greater than that of B. As in the configuration for *opt*, free movement of blocks between A and B is allowed in this scheme, provided that the contents of A and B are disjoint. Management is adapted from Belady's MIN algorithm, as follows. On a miss, the incoming block fills an empty slot in the corresponding set of cache A or cache B, if one exists. If no such empty slot exists, then for each set of cache A, an extended set of blocks is defined, consisting of all blocks in cache A and cache B that map to this set in cache A. For each extended set that includes at least one block currently resident in cache B, i.e. sets whose extended set is larger than the associativity of cache A, block γ of the extended set whose next reference is farthest in the future is found. If block γ does not currently reside in cache B, it is swapped with one of the cache B blocks of this extended set. The incoming block is placed in cache A and the block that in that set next referenced farthest in the future is moved to cache B, overfilling B by one block. The cache B block next referenced farthest in the future is then replaced. This choice is not optimal in all cases, as illustrated by the following example.

For the reference pattern in Figure 3.3, consider a design with a direct-mapped cache A of size 2 blocks (2 sets) and a one block cache B. The references shown in the figure are block addresses. Upper case letters map to set 0 and lower case letters map to set 1 of cache A. The figure shows the contents of set 0 and set 1 in cache A and cache B after

Reference	b	c	F	D	b	E	D	F	c
Time	1	2	3	4	5	6	7	8	9
Set 0	–	–	F	F	F	D	D	D	D
Set 1	b	b	b	b	b	b	b	b	b
Cache B		c	c	D	D	E	E	F	c
Hit/Miss	***M***	***M***	***M***	***M***	***H***	***M***	***H***	***M***	***M***

Figure 3.3: An example showing why *pseudo-opt* is suboptimal.

each memory access. Using *pseudo-opt* we incur 7 misses. The first 3 compulsory misses fill empty blocks. **D** replaces **c** at time 4 since **c** is next referenced farther in the future than **F** or **b.** At time 6, **E** cannot replace **b** and replaces **F** rather than **D**. Finally, **F** and **c** miss. However, the minimum possible number of misses is 6. This can be achieved by replacing **F** (in set 0 of cache A) instead of **c** (in cache B) at time 4. **E** then replaces **b** at time 6 after swapping **b** and **c**, and finally **F** misses, but **c** hits.

In cases where the A and B caches are fully-associative, *pseudo-opt* reduces to *opt*. Although *pseudo-opt* performance, as seen in Table 3.4, is close to that of *opt*, it is also not an implementable policy. Furthermore, *pseudo-opt's* performance is much better than the performance of the implementable multilateral schemes. Most of the performance difference between the implementable schemes and *opt* is thus due to non-optimal allocation and replacement decisions; only a small portion of the performance difference (no more than the difference between *opt* and *pseudo-opt*) is due to the restricted associativity of the implementable schemes. We therefore use *pseudo-opt* for comparison to the implementable schemes in order to eliminate the associativity difference from the evaluations and give a better idea of the realizable performance of a limited-associativity multilateral cache. The differences in placement and replacement decisions seen in the implementable schemes provide insights into their performance relative to a near-optimal scheme.

3.5.2 Simulation environment

To evaluate the *opt* and *pseudo-opt* schemes, we collected memory reference traces generated from the *SimpleScalar* processor environment [21]. Each trace entry contains the (effective) address accessed, the type of access (load or store), and the program counter of the instruction responsible for the access.

We skipped the first 100 million instructions (to avoid initialization effects) and analyzed the subsequent 25 million memory references. We limited the number of memory references evaluated due to the space and processing time required to perform the *opt* and *pseudo-opt* cache evaluations. These experiments use the five SPEC95 integer benchmarks — **compress**, **gcc**, **go**, **li,** and **perl,** since sampling such a small portion of a floating point program will likely generate references that form part of a regular loop, resulting in very low miss rates. However, these trace portions of the integer programs do reasonably mirror the actual memory reference behavior of the complete program execution as seen in Section 3.4.

The traces were annotated to include the information necessary to perform the *opt* and *pseudo-opt* replacement decisions, and counters for many useful statistics. These included the outcome of an access (hit or miss) as it would have occurred in an *opt/ pseudo-opt* configuration, the usage information for each block tour (as seen for each scheme), and the number of blocks in each reuse category that are in the cache at each instant in time (i.e. the number of NTNS, NTS, TNS, and TS blocks resident in the cache). From these statistics, we gathered information regarding the performance of the *opt* and *pseudo-opt* management schemes and compared the performance of the implementable schemes to that of the optimal schemes on an access-by-access basis.

Due to the smaller size of the input sets in this section, compared to the full program executions performed in Section 3.3, we chose a direct-mapped, 8K A cache and a fully-associative, 1K B cache, each with a 32B block size, for the *pseudo-opt* configuration. The *opt* configuration is simply a 9K fully-associative cache. Using the larger, (16+2)K caches of Section 3.3 for these evaluations would not have been useful in determining each scheme's performance, as the SPEC benchmarks already have small to moderately sized working sets [10]; these relatively short traces would show little performance benefit from active management in a large cache, whereas the benefits of active management are highlighted when using smaller caches. The *mlcache* simulator used in this section only deals with the cache and memory, not the processor. Without processor effects, timing is of little significance and *mlcache* is used here only as a behavioral-level simulator.

3.5.3 Results

3.5.3.1 Analysis of *opt* vs. *pseudo-opt*

We analyzed the annotated traces produced from the *opt* and *pseudo-opt* runs to determine their relative performance based on miss ratio measurement and block usage information. In particular, we counted the number of tours that show each reuse pattern, the number of each type of block resident in the cache at any given instant, and how often a block with a prior reuse characteristic changes its usage pattern in a subsequent tour.

3.5.3.2 Miss ratio

As the miss ratios in Table 3.4 show, the performance of *pseudo-opt* is relatively close to that of *opt*, except for **go**. Note that miss ratio in these experiments is a straightforward performance metric, as the simulations are behavioral and do not include

	Compress	Gcc	Go	Li	Perl
opt	0.042	0.013	0.008	0.008	0.0169
pseudo-opt	0.048	0.016	0.023	0.010	0.0174
PONS	0.050	0.018	0.028	0.011	0.0175
NTS	0.063	0.029	0.054	0.018	0.0200
PCS	0.067	0.032	0.060	0.019	0.0190
16K:1w	0.063	0.029	0.050	0.020	0.0520

Table 3.4: Miss ratios for the five (8+1)K and the 16K cache configurations on the trace inputs. *PONS* is the *pseudo-opt no-swap* scheme.

access latencies or processor latency-masking effects. In **go**, the performance disparity between *opt* and *pseudo-opt* is due to the limitations on the associativity of the A cache, and possibly also to the suboptimal replacement policy of *pseudo-opt.* On each replacement, any number of swaps can be done in *opt* to rearrange the cache contents so that the least desired cache block is replaced. However, in *pseudo-opt*, the movement choices are limited by the mapping requirements of the direct-mapped A cache — at most one block, mapping to the set in B that is associated with the incoming block, may be swapped on each replacement. Despite its more limited choice of blocks to replace, the performance of *pseudo-opt* is still very close to *opt* except for **go**, and the difference in performance between *pseudo-opt* and *opt* is always much smaller than the difference between the implementable configurations and *pseudo-opt*. The actual performance of the implementable schemes is discussed in Section 3.5.4.

In addition to the advantage of future knowledge, *pseudo-opt* differs from the implementable schemes by freely allowing blocks to move between the A and B caches in order to obtain the best block for replacement. However, this movement actually accounts for very little of the performance difference seen between *pseudo-opt* and the implementable schemes.

To verify this claim, we created a version of *pseudo-opt*, called *PONS* (*pseudo-opt no-swap*), that disallows data movement between the caches. The management scheme used by *PONS* is the same as for *pseudo-opt*, except that no blocks are swapped between cache A and B at any time. The incoming block simply replaces the block in its A or B set that is next referenced farthest in the future. At a given time, fewer replacement candidates are available in *PONS* than in *pseudo-opt*. When a miss occurs, *PONS* considers the set in A and the set in B that the miss block maps to and replaces the currently resident block next referenced farthest in the future; however, *pseudo-opt*, which can freely swap blocks between A and B, also considers other sets in A that map to this set in B. Due to its more restricted choice of blocks for replacement, the performance of *PONS* will never exceed that of *pseudo-opt*.

However, we see in Table 3.4 that the miss ratios of *pseudo-opt* and *PONS* are actually very close, indicating that the omission of inter-cache data movement has a small effect on near-optimum performance. In particular, the performance difference between *pseudo-opt* and *PONS* is much smaller than the difference between *pseudo-opt* and the implementable schemes, showing that the major advantage of *pseudo-opt* comes from its management using future knowledge as opposed to its ability to move blocks between caches. Since some multilateral schemes do allow data movement between the A and B caches, we will continue to use *pseudo-opt*, rather than *PONS*, as the basis for comparison to the implementable multilateral cache schemes.

3.5.3.3 Cache block locality analysis

We examined the locality of cache blocks that are resident in the cache structure at

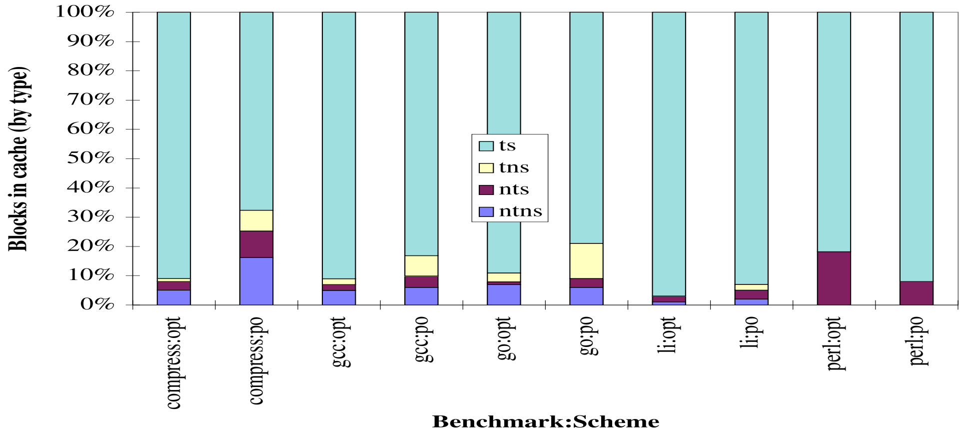


Figure 3.4: Dynamic cache block occupation in *opt* and *pseudo-opt* (denoted *po*), grouped into NTNS, NTS, TNS, and TS usage patterns.

any given time by counting the number of blocks in each category at the time of each miss and taking an average over the duration of the program. The locality of the cache blocks for the *opt* and *pseudo-opt* configurations is shown in Figure 3.4.

For *opt* and *pseudo-opt* managed caches, as expected, data that exhibits temporal reuse occupies a large portion of the cache space; nontemporal data occupies at most 23% of the cache (**compress** under *pseudo-opt*). Furthermore, the vast majority of the blocks in L1 are both temporal *and* spatial. Though the blocks are relatively small in size (32 bytes), we find that spatial reuse can be exploited well if the cache is managed properly. As expected, the *pseudo-opt* configuration (vs. the *opt* configuration) generally holds fewer TS blocks in the cache, due to the *pseudo-opt* configuration's limited placement options and sub-optimal replacement decisions. However, **perl** is an exception to this observation. In both *opt* and *pseudo-opt*, all of **perl's** data in the cache is spatial (TS or NTS), as seen in Figure 3.4; keeping a nontemporal spatial block in cache long enough so that it obtains a temporal spatial status, as done by *pseudo-opt*, slightly increases the miss ratio of the **perl** benchmark above that of *opt*, indicating that maximizing the number of TS blocks in cache is not always the best policy.

3.5.3.4 Block usage persistence

While the presence of certain types of blocks in the cache shows the potential benefit of managing the cache with reuse information, management based on this information is not straightforward. Blocks can have different usage characteristics during different portions of program execution, making block usage hard to predict. The prediction of a particular block's usage pattern is similar to the branch prediction problem. However, branch outcomes are easier to predict correctly than optimal block usage characteristics for aiding placement decisions.

To assess the value of reuse information, we examined the *persistence* of cache block behavior in the *opt* and *pseudo-opt* schemes, i.e. once a block exhibits a given usage characteristic in a current tour, how likely is it to maintain that characteristic in its next tour? If there is a high correlation between past and future use (i.e. the block's usage characteristic is persistent), prediction of future usage behavior will be easier. Block *persistence* in terms of *same* and *not-same* is therefore analogous to the *same-direction* terminology used in branch prediction studies [34] to predict the path a specific branch will take given behavior history. Instead of determining block persistence in terms of the four usage patterns NTNS, NTS, TNS, and TS, we decided to examine only the persistence of T and NT patterns. This coarser granularity grouping is more directly relevant to our placement decisions for 2-unit cache structures.

Figure 3.5 presents data for block usage persistence in successive tours. In general, blocks that exhibit NT usage behavior in prior tours have a strong likelihood of exhibiting NT behavior again in future tours. In the *opt* scheme, this likelihood ranges from 63%

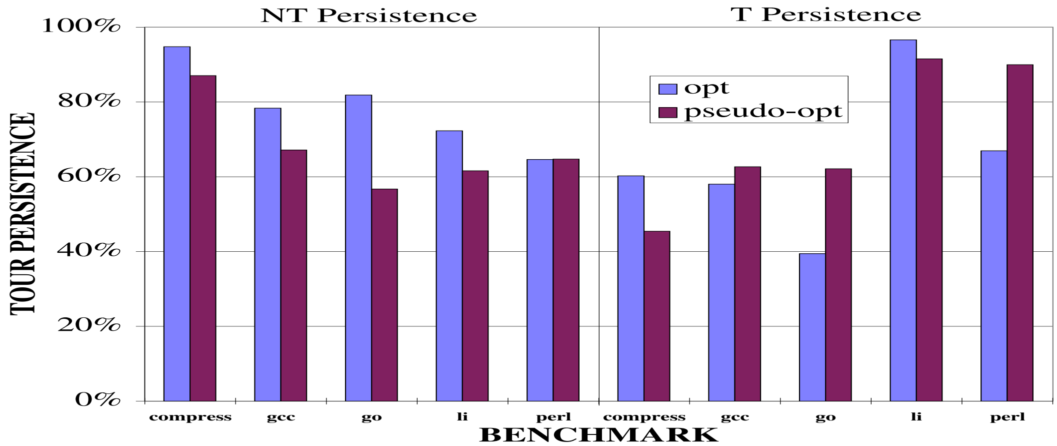


Figure 3.5: Block usage persistence within *opt* and *pseudo-opt*.

(**perl**) to 95% (**compress**) for the evaluated benchmarks. However, the *pseudo-opt* scheme shows somewhat less persistence, ranging from 57% (**go**) to 87% (**compress**) for NT blocks. Overall, the persistence of NT blocks in both *opt* and *pseudo-opt* schemes is well over 50%. Blocks that exhibit T usage behavior are less persistent, and thus less predictable. T block persistence in the *opt* scheme ranges from 40% (**go**) to 97% (**li**), and is similar for *pseudo-opt*: 45% (**compress**) to 92% (**li**).

li skews these numbers somewhat; regardless of the type of usage characteristics that a block exhibited in a tour, its next tour is highly likely to exhibit temporal reuse. While temporal blocks are persistent in **li**, they are much less persistent for the other four benchmarks. As **compress**, **gcc**, **go,** and **perl** represent a wider range of program execution than **li** alone, we see that future tours of temporal blocks are harder to predict than nontemporal blocks, i.e. temporally tagged blocks are only weakly biased toward exhibiting T usage patterns in their next tour.

3.5.4 Multilateral scheme performance

Given the performance and reuse information of the *opt* and *pseudo-opt* configurations, we can determine how the implementable schemes perform as a result. We restrict our evaluation of the implementable schemes to NTS and PCS, the two multilateral configurations that actively place data within L1 based on individual block reuse information.

3.5.4.1 Miss ratio performance of NTS and PCS

To compare with the *pseudo-opt* configuration used, we configured the NTS and PCS structures to have an 8K direct-mapped A cache, a 1K fully-associative, LRU-managed B cache, 32B block size, and a 32-entry detection unit (DU).

The miss ratio performance of the two configurations is shown in Table 3.4 along with the performance of *opt*, *pseudo-opt*, *PONS*, and a direct-mapped 16K single structure cache. In concurrence with results of earlier analyses of these multilateral configurations [6][7][32], the NTS and PCS caches each perform about as well as a direct-mapped cache of nearly twice the size. However, as the performance of *pseudo-opt* and *PONS* indicates, there is still room for further improvement.

3.5.4.2 Prediction accuracy

NTS makes placement decisions based on a given block's usage during its most recent past tour; if the block exhibited nontemporal reuse, it is placed in the smaller B cache for its next tour, otherwise it is placed in the A cache. PCS makes placement decisions based on the reuse of blocks loaded by the memory accessing instruction. If the most recently replaced block loaded by a particular PC exhibited nontemporal reuse, the next block loaded by that PC is predicted to do the same and is placed in the smaller B cache, otherwise it is placed in the larger A cache. In both schemes, accessed blocks with no matching entry in the DU are placed in the A cache by default.

The accuracy of these predictions can be determined based on the actual usage of the blocks in the *pseudo-opt* scheme. For instance, a prediction of T behavior for a block is classified as correct if the actual usage of that block in the *pseudo-opt* scheme is T. As noted in Section 3.5.2, the annotated traces fed to our trace-driven cache simulator, a modified version of *mlcache*, contain the actual block usage information for the tours seen in the *pseudo-opt* scheme. The simulator provides information on the selected scheme's block prediction and actual usage accuracy. Given this information, it is easy to determine how well each of the configurations predict a block's usage, and consequently, whether it is properly placed within the L1 cache structure.

Table 3.5 shows the prediction accuracy of NTS and PCS for the benchmarks examined. In general, ignoring **li** due to its excessively high temporal reuse, the prediction accuracy is relatively low, ranging from 25% (**compress** with NTS) to 60% (**go** with PCS), despite the larger granularity of block typing (two categories, T and NT, rather than the complete four category breakdown). Both the NTS and PCS schemes show poor prediction accuracies, directly impacting their block allocation decisions and their resulting overall performance. Improved block usage prediction for these schemes may result in better block placement and higher performance.

3.5.4.3 Actual usage accuracy

Regardless of the prediction accuracy, a given block will exhibit reuse characteristics based on the duration and time of its tour through the cache. We examined the actual usage of each block as it was evicted from each of the caches in the implementable schemes to see how that usage compared to the same block's usage in the *pseudo-opt* scheme. This comparison sheds some light on the effect of eliminating the

		Tour prediction accuracy		Actual usage accuracy	
		NTS	**PCS**	**NTS**	**PCS**
Compress	**NT**	0.483	0.703	0.859	0.828
	T	0.240	0.210	0.402	0.412
	Total	*0.247*	*0.495*	*0.725*	*0.725*
Gcc	**NT**	0.322	0.589	0.684	0.617
	T	0.539	0.596	0.703	0.704
	Total	*0.527*	*0.595*	*0.696*	*0.675*
Go	**NT**	0.337	0.454	0.573	0.519
	T	0.588	0.621	0.646	0.652
	Total	*0.561*	*0.596*	*0.624*	*0.609*
Li	**NT**	0.225	0.243	0.390	0.363
	T	0.815	0.815	0.873	0.866
	Total	*0.786*	*0.723*	*0.787*	*0.758*
Perl	**NT**	0.406	0.817	0.934	0.950
	T	0.326	0.382	0.727	0.738
	Total	*0.327*	*0.510*	*0.861*	*0.875*

Table 3.5: Tour prediction and actual usage accuracy for NTS and PCS, broken into NT, T, and overall (total) accuracy. Accuracies are relative to actual block usage in *pseudo-opt*.

movement of blocks between the A and B caches during an L1 tour.

We see in Table 3.5 that relative to the low prediction accuracy we saw in Section 3.5.4.2, the actual usage of the blocks in the cache is closer to the actual usage of the blocks in *pseudo-opt*. Thus, despite some poor placement decisions and restricted movement, some blocks still exhibited reuse behavior akin to that seen in the *pseudo-opt* configuration, where free movement between A and B is allowed. Furthermore, although NTS exhibited lower tour prediction accuracy than PCS (in all the benchmarks except for **li**), it exhibited higher actual usage accuracy (in all the benchmarks except for **perl**).

3.5.4.4 Tour lengths

While these accuracies are interesting, the block tour lengths in the implementable schemes are not directly related to the tour lengths in the *pseudo-opt* scheme, as *pseudo-opt* has a more elaborate replacement policy. Disparate tour lengths can affect these comparisons in two ways. First, for tours that are shorter in the *pseudo-opt* configuration than in the implementable schemes, the implementable schemes may keep a block in L1 longer than necessary and permit it to exhibit seemingly more beneficial usage patterns (i.e. TNS or TS). While this makes a particular block seem more useful, the longer presence of that block in L1 in the implementable schemes may unduly shorten the tours of a significant number of other blocks, leading to their misclassification and precluding their optimal placement.

Conversely, where tours are longer in the *pseudo-opt* configuration than in the implementable schemes, the corresponding blocks in the implementable schemes may be replaced before they can exhibit their optimal usage characteristics, causing poor usage accuracy, poor prediction accuracy, and improper placement into the small B cache, and hence shorter tour lengths.

Table 3.6 shows the average tour length for blocks showing T and NT usage characteristics. As expected, NT tours are much shorter than T tours, and tours in *pseudo-opt* are on average much longer than tours in either NTS or PCS, due to its future knowledge and greater flexibility in management of data once it has entered the L1 cache structure. In NTS and PCS, once a block is placed in L1, it remains in the same cache until it is replaced, and is thus subject to the replacement policy inherent in the corresponding cache. For instance, in either NTS or PCS, if block α is deemed to be T it is placed in the direct-mapped A cache. If a subsequent block β maps to the same set in A and is also

	Compress		Gcc		Go		Li		Perl	
	NT	T	NT	T	NT	T	NT	T	NT	T
p-o	1293	26695	1731	35946	1179	22297	4400	34222	957	54749
NTS	4.1	44.6	2.0	52.0	1.4	26.1	3.3	66.3	7.7	130.4
PCS	3.7	48.9	2.1	49.8	1.3	24.1	3.2	66.1	7.9	134.0
8k:1w	3.6	35.4	2.0	35.7	1.2	14.0	2.6	48.0	3.7	35.9

Table 3.6: Average tour lengths for *pseudo-opt (p-o),* NTS, PCS, and a direct-mapped single structure cache. Tour lengths are measured by the number of accesses handled while the block is in cache. Tours are broken into NT and T.

marked T, the earlier T block, α, will be evicted, possibly before it can exhibit its optimum usage characteristics. In *pseudo-opt,* if it is deemed desirable at that time, block α would simply be moved to the B cache. As a result, the tour length of block α would tend to be much longer under *pseudo-opt* than under NTS or PCS.

We see that there is a clear gap between the average tour length of blocks in *pseudo-opt* vs. the implementable schemes. This difference in tour lengths is, however, much larger than the difference in miss ratio for the implementable vs. *pseudo-opt* schemes. The performance difference is much smaller because, although the average tour length in *pseudo-opt* has increased greatly, the tour lengths of conflicting blocks may not be helped as much by the better management scheme. Some blocks may be chosen to remain in cache for nearly the entire program's execution, as they are accessed regularly (though not necessarily frequently), and would thus have very long tour lengths. These long-lived blocks greatly increase the average tour lengths seen for each benchmark, though they may only reduce the overall number of misses by a small amount.

From this study we see that by improving upon the prediction of the usage characteristics of a block and the management of those blocks once they are placed in the

L1 cache structure, we might reduce the performance gap that exists between these schemes and *pseudo-opt*, and hence improve the performance of NTS and PCS. We see from Table 3.6 that in every case, each of the multilateral schemes does have longer tour lengths relative to the single, direct-mapped cache structure of nearly the same size, indicating that the use of the additional B cache in the multilateral schemes does allow blocks to remain in L1 longer.

3.6 Summary

In this chapter we have evaluated three different implementable methodologies (MAT, NTS, and PCS) for managing an on-chip data cache based on active block allocation via capturing and exploiting reuse information. In general, an actively managed cache structure significantly improves upon the performance of a traditional, passively-managed cache structure of similar size and competes with one of nearly twice the size. Further, the individual effective address reuse history scheme used in NTS generally gives better performance than the macroblock effective address-based MAT or the PC-based PCS approaches.

We compared the performance of the NTS and PCS schemes to the performance of a near-optimally managed cache structure (*pseudo-opt*). The difference in performance, block usage prediction, actual block usage, and tour lengths between the implementable schemes and *pseudo-opt* shows that the actively managed multilateral caches still have much room for improvement.

Thus, multilateral cache structures that actively place data within the cache show promise of improving cache space usage. However, the prediction strategies used in these current schemes are too simple. Improving the prediction algorithms, as well as actively managing blocks once they are placed in the L1 cache structure (active replacement), can help improve the performance of the implementable schemes and may enable them to more nearly approach optimal cache space usage.

CHAPTER IV

ALLOCATION BY CONFLICT: A SIMPLE, EFFECTIVE MULTILATERAL ALLOCATION SCHEME

4.1 Introduction

Many reuse-based multilateral schemes have been proposed, including NTS [3][6] and PCS [6]. While these multilateral cache schemes have been shown to perform well (often equalling or exceeding the performance of traditional single structure caches nearly twice as large), their performance is still significantly short of optimal (see Chapter III). Their shortcomings are due to their suboptimal allocation decision mechanisms and their lack of active replacement. In particular, NTS and PCS make their allocation decisions based on the past tour usage of incoming blocks, though block usage is not highly persistent from one tour to the next [37][38]. Complicating matters further, a Victim scheme [18], which uses no past tour reuse information in its management, has been shown to outperform these reuse-based allocation schemes when the A cache is direct-mapped [6].

As a result, we have developed a new multilateral allocation scheme, Allocation By Conflict (ABC), in the hopes of outperforming Victim when A cache associativity is low and competing comparably with or outperforming NTS and other proposed reuse-based allocation schemes. ABC makes its allocation decisions based on the current usage of blocks within the cache, as opposed to past tour usage of blocks or instructions, as in NTS and PCS, respectively. Put simply, ABC allocates a block to the A cache if the LRU element of its set in A has not been reaccessed since the last conflict to that set; otherwise, the block is allocated to the B cache. ABC makes active replacement decisions based on whether to retain or replace the candidate block in A as a function of its current tour behavior since the last conflict to its set. ABC's performance gains derive from its ability to reduce the impact of conflict misses and its ability to remove some other block from the cache when the LRU element is deemed to still be useful.

In this chapter we discuss the ABC scheme operation in detail and compare the performance of ABC to NTS [3], PCS [6], and Victim [18]. To provide a performance baseline for comparing multilateral structures of a given size and configuration, we also evaluate the performance of a Random allocation scheme. We find that the ABC scheme performs best among the reuse-based allocation strategies, and that ABC even performs comparably to Victim when the A cache associativity is low. Interestingly, the Random scheme's performance is also good for such configurations, despite its lack of "intelligent" decision making. As the associativity of the A cache increases, the reuse-based schemes (NTS, PCS, and ABC) all continue to sustain their performance gains, while the gains offered by Random and Victim degrade.

4.2 A new multilateral allocation scheme

Given the apparent shortcomings of the previously evaluated multilateral schemes and the good performance of Victim (when the A cache is direct-mapped), the merit of using past tour reuse information in multilateral cache management is somewhat questionable. Therefore, we developed a new multilateral allocation scheme, Allocation By Conflict (ABC), which challenges the performance of the previously evaluated schemes. The ABC scheme makes allocation decisions based on the current usage of data in the cache. In particular, ABC focuses on reducing the impact of conflict misses by prolonging the tour lengths of LRU, but still actively referenced blocks in the A cache. When a miss occurs, ABC focuses on the *conflict block*, i.e. the LRU block in the set of A that the miss block maps to — namely, the block that the miss block would replace if it were allocated into A. The conflict block is allowed to remain in A unless 1) at least one conflict to its set in A has occurred during its tour in A, and 2) this conflict block has not been referenced since the last such conflict. When the conflict block remains in A, the miss block is simply allocated into B, replacing the LRU block in the set it maps to in B. For conciseness, we refer to the set of A to which the miss block maps as the *A-set* and the set of B to which the miss block maps as the *B-set*. Thus, in contrast with the other active management schemes for multilateral caches, ABC uses an active *replacement* strategy for the A cache, rather than an active *allocation* strategy.

We also discuss the Random allocation scheme, which is used in this chapter to set a performance baseline for multilateral caches of a given size and configuration.

4.2.1 Allocation By Conflict

One of the main benefits of multilateral caches is the ability to reduce the performance impact of conflict misses. Multilateral caches can allow the number of blocks that map to a particular set in A to exceed the associativity of A and reside in L1 concurrently. The Victim cache reduces pressure on these "hot" cache sets by saving A cache replacement victims in the smaller B cache. The NTS and PCS caches indirectly reduce conflicts by allocating a miss block to B according to its past tour reference behavior or the past tour reference behavior of blocks related (through the PC) to the miss block. Although the NTS/PCS allocation mechanisms are based on the past tour usage of the incoming miss block or related blocks, allocating the miss block to B can allow it to reside in L1 concurrently with a conflicting block currently resident in A, thereby prolonging the tour length of the conflict block.

The Allocation By Conflict (ABC) multilateral allocation scheme is also based on the fundamental principle of using a multilateral cache for conflict reduction. Like the NTS and PCS schemes, caches A and B are placed equidistant from the CPU, and there is no data path between the two caches. However, unlike NTS/PCS, ABC makes its block allocation decisions based on whether to replace the conflict block in A, instead of what the predicted usage of the incoming miss block might be.

In the ABC scheme, the LRU element of a miss block's A-set, α, is replaced unless it has been referenced since the last conflict to this A-set. If α has not yet experienced a conflict during this tour or if it has been accessed since the last conflict to its A-set, the incoming block is allocated to B, replacing the LRU element in its B-set. These decisions allow blocks allocated to A to survive at least one conflict to their set before being evicted. Furthermore, if a block is accessed at least once between successive conflicts to its A-set, it can remain in A indefinitely. If α has not been accessed since the last conflict to its A-set, it could have been replaced during that last conflict without harming the cache state from then until now; it is therefore considered to be a good candidate for replacement now. Thus, ABC makes active replacement decisions based on whether to retain or replace the conflict block in A. Furthermore, it makes this decision as a function of the current tour behavior of the conflict block since the last conflict to its set. By contrast, NTS and PCS make their active allocation decisions based on the past tour characteristics of the miss block or miss instruction, respectively.

Each block in cache A is augmented with a single bit, the Conflict bit (C). The `C=0` state indicates that either no conflicts to its A-set have occurred during the block's current tour or the block has been accessed since the last conflict to its A-set. If a miss occurs while the LRU block in the A-set (the conflict block) has `C=0`, then the miss block is allocated to B and the conflict block is thereby retained in A. The `C=1` state indicates that a conflict to its A-set has occurred during the current tour, but this block has not been reaccessed since the last such conflict. If a miss occurs while the conflict block has `C=1`, the conflict block in A is replaced by the miss block.

When a block X is brought into cache, if its target set in the A cache is not full, X is simply allocated to the larger A cache. When its target set in A is full, the ABC allocation mechanism is put into effect. Let α be the conflict block of the A-set, and β be the conflict block of the B-set. (Note that α and β may refer to different blocks in L1 at different points in time.) If the Conflict bit of α is equal to 0, X is allocated into B, causing β to be replaced. Whenever a miss block is allocated into B, the Conflict bit of every block in its A-set is set to 1 to indicate that a conflict has occurred for that set.

Whenever a block in A is accessed (an A cache hit), its Conflict bit is reset to 0. If a subsequent miss block Y maps to that set in A and `C=0` for α, Y is placed in B and β is replaced. This decision is made because α has shown that it has been used since the last conflict to the set has occurred (or, in the base case, that no conflict has yet occurred during α's current tour), and α is thus deemed to have a good likelihood of being accessed again in the near future. When Y is placed in B, the Conflict bit of the A-set elements are set to 1 (`C=1`), so that the A cache resident blocks must "prove" themselves again in order to remain in L1 past the next conflict to this set.

If a subsequent miss block Z maps to that set in A and `C=1` for α, this indicates that α has not been used since X (or Y) was allocated in L1. Therefore, it is likely that α will not be used again soon and it could have been replaced by X or Y without harm to the cache state since the allocation of X (or Y). Thus, Z is allocated to A, replacing α. The Conflict bit of Z is set to 0 so that the new block has a chance to remain in A for at least a short period of time; otherwise, another conflicting block may miss and cause this new block to be evicted before it can be reused.

4.2.2 Random allocation

The Random allocation scheme has no path between the A and B caches and uses LRU replacement in each cache; however, blocks are allocated by simply using a "virtual coin-flip" to decide whether the block is allocated to A or B. Even though this scheme uses no intelligence in its allocation decisions, Random does provide a remarkably good performance baseline for a multilateral cache of a given size and configuration.

Random requires some sort of random number generator that provides a uniform distribution of outputs so that half of the incoming blocks are allocated to A, and the other half to B. This randomness can be obtained any number of ways, e.g. a random number generator, sampling the real-time clock, etc. This 50/50 split of allocations could be adjusted depending upon the configurations of the A and B caches used, but in the performance comparisons that follow, Random always utilizes the 50/50 split. We evaluate the performance of different allocation ratios (i.e. 40/60, 60/40, etc.) in Section 4.5.3 and show that 50/50 is in fact a good choice for the configurations studied.

	Single	Victim		ABC/NTS/PCS/Random	
Cache	A	A	B	A	B
Size	8/16K	8K	1K	8K	1K
Associativity	1,2,4,8/1,2,4	1	full	1	full
Replacement Policy	–,LRU,LRU,LRU/ –,LRU,LRU	–	LRU	–	LRU
move time	–	0/2/4		–	
latency to next level	18	18	–	18	18

Table 4.1: Characteristics of the six configurations studied. Times/ latencies are in cycles.

4.3 Simulation methodology

To evaluate the performance of the new multilateral allocation schemes, ABC and Random, we perform two main experiments, one using execution-driven simulation and the other using trace inputs. For the execution-driven evaluations, we used the same processor and cache (*SimpleScalar+mlcache*) setup as described in Section 3.3. The cache schemes we chose to evaluate are: NTS, PCS, Victim, ABC, and Random, along with various single structure caches; the evaluated configurations are summarized in Table 4.1. We simulate both (8+1)K and (16+2)K multilateral cache configurations, but focus our analysis on the former as they show more of the benefits of active management than do the larger cache structures evaluated in Section 3.3. We expanded our benchmark suite to include all of SPEC95; the characteristics of the SPEC95 benchmarks are described in Table 4.2. As in Section 3.3, the test inputs were used and each benchmark was run to completion or through the first 1.5 billion instructions.

We also perform a short performance comparison in Section 4.4.4 using the benchmarks, configurations, and traces from Section 3.5. Using these traces, we can see how ABC, Random, and Victim perform relative to near-optimally managed structures of

Program	Instruction Count (millions)	Memory References (millions)		Perfect Cache Performance	
		Loads	Stores	Cycle Count (millions)	IPC
SPEC95 Integer Benchmarks					
Compress	35.68	7.37	5.99	5.35	6.6644
Gcc	263.85	61.15	36.24	43.50	6.0648
Go	548.13	115.79	41.40	91.33	6.0049
Ijpeg	1,046.09	277.71	101.21	144.15	7.2572
Li	956.49	286.38	168.79	151.32	6.3210
M88ksim	413.28	85.64	41.83	72.48	5.7020
Perl	1,500.00	396.82	269.83	232.89	6.4408
Vortex	1,500.00	445.95	355.69	196.83	7.6206
SPEC95 Floating Point Benchmarks					
Applu	1,500.00	311.86	70.29	184.08	8.1487
Apsi	1,500.00	341.57	127.37	288.64	5.1969
Fpppp	1,500.00	576.25	220.37	201.09	7.4595
Hydro2d	974.50	196.11	60.90	127.63	7.6353
Mgrid	1,500.00	528.07	21.53	194.84	7.6985
Su2cor	1,054.09	262.20	84.74	152.34	6.9192
Swim	849.92	205.18	58.44	113.02	7.5201
Tomcatv	1,500.00	316.25	176.02	321.75	4.6621
Turb3d	1,500.00	207.24	132.10	235.68	6.3644
Wave5	1,500.00	321.87	133.26	318.69	4.7067

Table 4.2: The 18 benchmarks and their memory characteristics.

similar size.

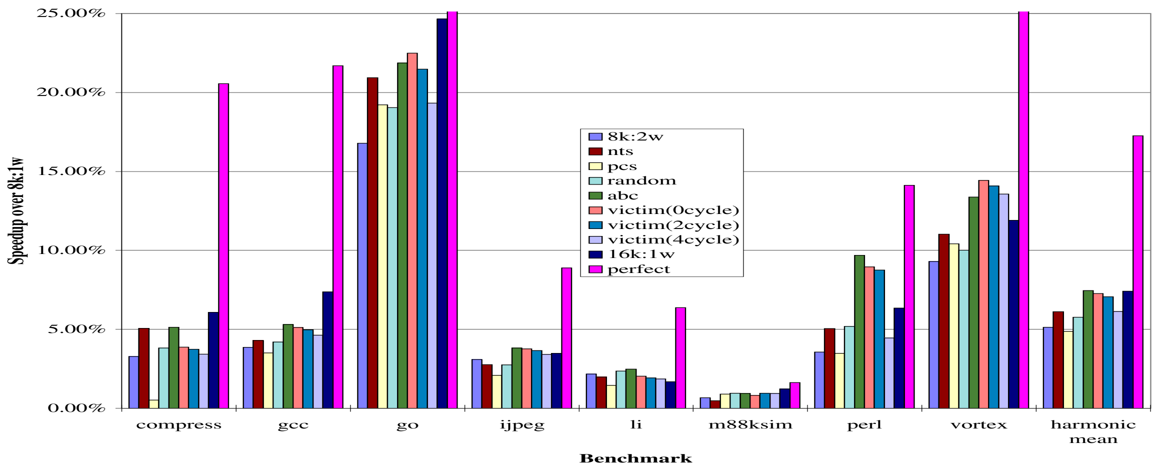
4.4 Experimental results

In this section, we present the performance of the direct-mapped A cache and fully-associative B cache (8+1)K multilateral cache configurations vs. single structure direct-mapped and two-way associative caches of similar size (8K) and also a single structure direct-mapped cache of nearly twice the size (16K). We then present the performance of the multilateral schemes as the associativity of the A cache is increased from direct-mapped to 2- and 4-way set associative. We compare the 2-way associative multilateral caches to 4-way 8K and 2-way 16K caches, and compare the 4-way associative multilateral caches to 8-way 8K and 4-way 16K caches.

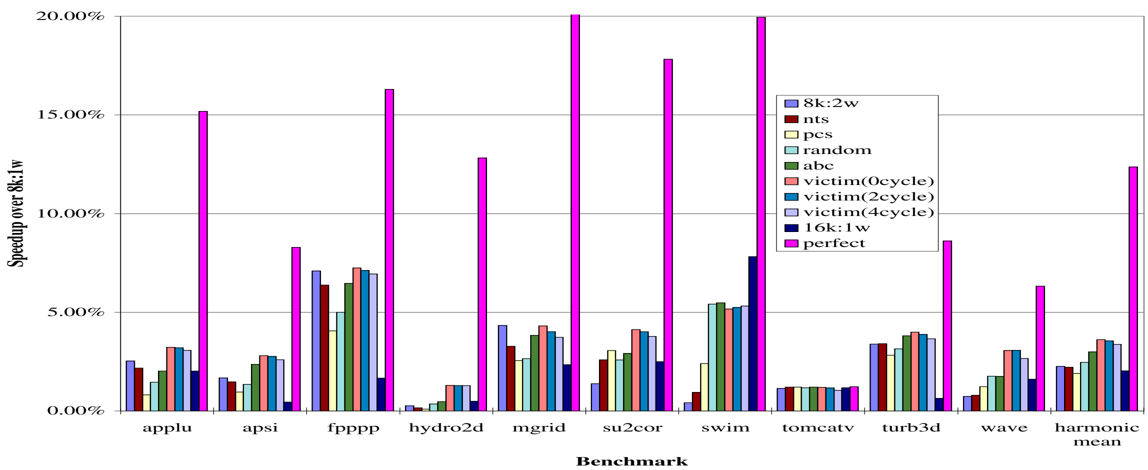
4.4.1 Direct-mapped A cache performance

The speedup achieved by each scheme for each program is shown in Figure 4.1, where the single 8K direct-mapped (8k:1w) cache is taken as the base. Overall, the speedup obtained by using the multilateral cache schemes ranges from virtually none in **hydro2d** to ~22% in **go** with Victim (0 cycle swap latency). Clearly, some of the benchmarks tested do not benefit significantly from any of the improvements offered by the cache schemes evaluated, i.e. better management of the L1 data store by the multilateral schemes, increased associativity of a single cache (8k:2w), or a larger cache (16k:1w). In benchmarks where there is appreciable performance gain over the base cache, the multilateral schemes often perform as well as or better than either a higher-associative single cache or a larger direct-mapped cache. However, in **compress**, **gcc**, **go**, and **swim**, the benchmark's larger working sets do benefit most from the larger overall cache space provided by the 16K direct-mapped structure, although even for these benchmarks the multilateral schemes are able to obtain a significant part of the performance boost via their better management of the smaller multilateral cache.

Among the multilateral schemes, we see several trends that were exposed in earlier studies [6][7][28][37]. With a direct-mapped A cache, the NTS scheme generally outperforms the PCS scheme (in all benchmarks except **m88ksim**, **su2cor**, **swim**, and **wave**), and the Victim scheme generally outperforms either NTS or PCS (the exceptions



i) SPECint95



ii) SPECfp95

Figure 4.1: Overall execution speedup for the six evaluated cache schemes, relative to a single direct-mapped 8K cache (8k:1w). SPECint95 performance is shown on top, SPECfp95 below. The three Victim results show the effect of increasing the swap latency between the caches. Perfect is the performance of a perfect cache. The harmonic mean speedup over the SPECint and SPECfp benchmarks is also shown. The Perfect speedup for go is 41.86%, vortex 33.8%, mgrid 20.4%, and swim 20%.

---

being **compress** and **tomcatv**, where NTS is best). Even when a swap latency is incorporated for the Victim scheme (either 2 or 4 cycles, as opposed to the base 0 cycle (free) swap), Victim outperforms NTS and PCS. The implications of swaps and swap latency in the Victim scheme will be discussed in depth in Section 4.5.1. However, when the associativity of the A cache increases, the performance of Victim decreases relative to

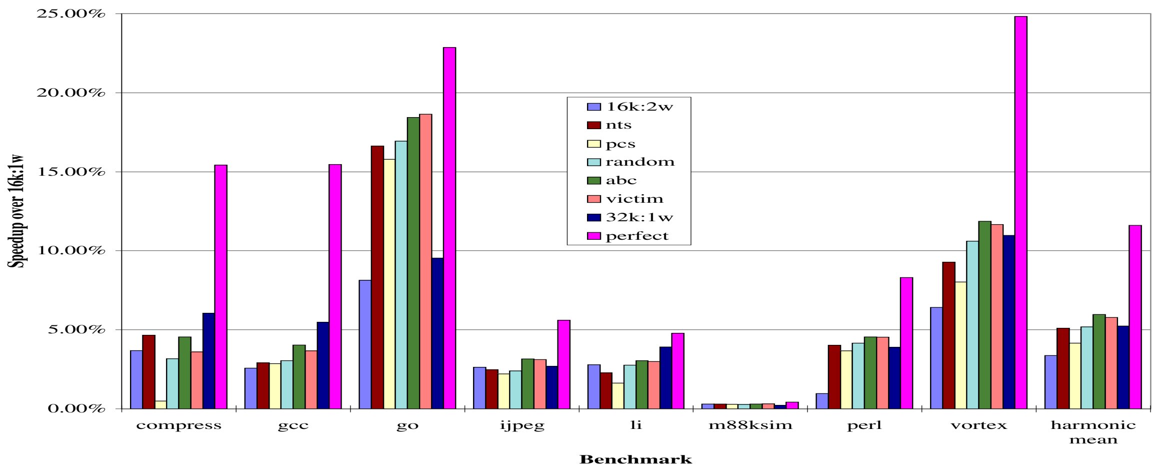
NTS and PCS, as we shall see in Section 4.4.2.

Interestingly, the Random allocation scheme also performs very well, equalling or beating both NTS and PCS in nearly half of the benchmarks (**ijpeg**, **li**, **m88ksim**, **perl**, **hydro2d**, **swim**, **tomcatv**, and **wave**) and in the harmonic mean of SPECfp, although NTS does outperform Random in the harmonic mean of SPECint. Random's performance is intriguing since, unlike NTS and PCS, it does not take into account any reuse information when making its allocation decisions. The question of the usefulness of reuse information as used by NTS and PCS in cache management will be explored further in Section 4.5.2.

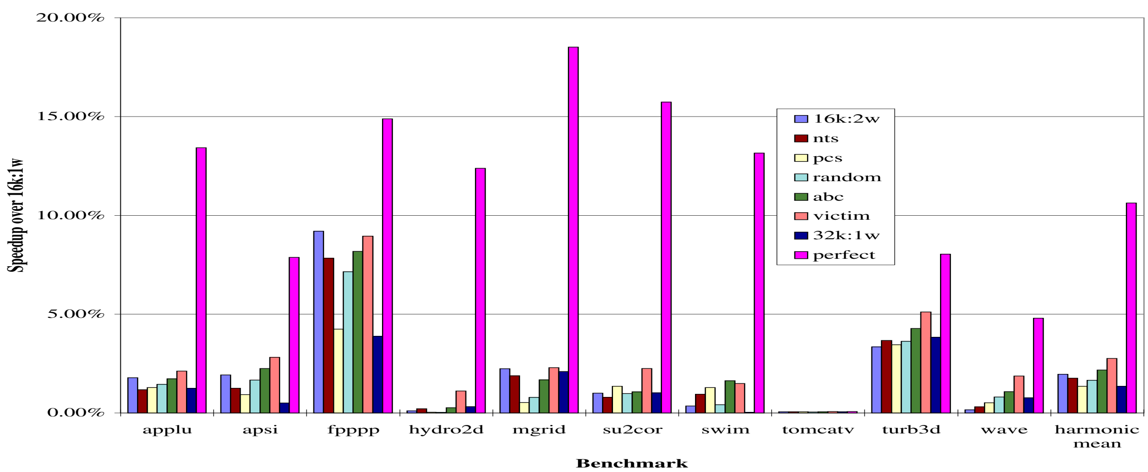
ABC is the top performing multilateral cache scheme among those that do not allow data movement between the subcaches (i.e. among NTS, PCS, Random, and ABC), and even outperforms Victim (even with 0 cycle swap latency) in all of the integer benchmarks except **go** and **vortex**. In the floating point benchmarks, ABC's performance lags Victim's, though as the swap latency increases, Victim's performance drops and becomes more comparable to ABC's performance. The performance of the ABC scheme vs. the Victim scheme will be further explored in Section 4.5.1.

Looking at the harmonic mean speedups, both the ABC and Victim caches equal (SPECint) or outperform (SPECfp) an associative cache of similar size (8k:2w) and a single structure cache nearly twice as large (16k:1w). In the case of the ABC cache, these performance gains are attained using very little additional hardware. The best harmonic mean speedups over the base 8k:1w cache are 7.45% (ABC) over the integer benchmarks and a more modest 3.61% (Victim with 0 cycle swap latency) over the floating point benchmarks.

The results scale well as larger cache sizes are used (and the working sets of the benchmarks do not fit in the cache). In Figure 4.2, we see the performance speedup for



i) SPECint95



ii) SPECfp95

Figure 4.2: Overall execution speedup for the six evaluated cache schemes with 16K direct-mapped A and 2K fully-associative B caches, relative to a single direct-mapped 16K cache (16k:1w). SPECint95 performance is shown on top, SPECfp95 below. Perfect is the performance of a perfect cache. The harmonic mean speedup over the SPECint and SPECfp benchmarks is also shown. Note: Victim shown is best case (0 cycle swap latency).

(16+2)K multilateral cache structures (i.e. 16K direct-mapped A cache plus a 2K fully-associative B cache) vs. 16K direct-mapped, 16K 2-way, and 32K direct-mapped single structure caches. Note that in this and the remaining experiments of this section, we present only the best-case Victim cache, with a 0 cycle swap latency. The performance trends mirror those found for the (8+1)K multilateral caches vs. a single structure 8K direct-mapped cache: once again, ABC is the best performing scheme among the SPECint benchmarks and Victim performs best overall for the SPECfp benchmarks. With the larger cache sizes, the 32k:1w cache outperforms the multilateral schemes in **li** (with the smaller cache sizes, the larger direct-mapped structure did not fare as well). Conversely, the 32k:1w cache shows no performance gain in **swim**, while the multilateral schemes still do. The overall performance gains over the base cache, however, are generally smaller now, as the base cache is able to contain more of the active working set. In fact, in **m88ksim** and **tomcatv**, the base cache already has performance almost identical to that of a perfect cache, and better cache structures and management policies cannot improve upon it. However, the remaining benchmarks do still show the performance benefits of using a given multilateral scheme instead of an associative cache of the same size (16k:2w) or a direct-mapped structure of nearly twice the size (32k:1w).

4.4.2 Associative A cache performance

The speedups achieved by each scheme as the associativity of the A cache is increased to 2-way and 4-way are shown in Figure 4.3 and Figure 4.4, respectively. The base configuration in each case is the single 8k:1w cache, so that we may compare the performance of the associative A cache multilateral caches to the direct-mapped A cache multilateral caches shown in Figure 4.1.

For a 2-way cache, the speedup obtained by using the multilateral cache schemes ranges from virtually none in **hydro2d** to ~29% in **go** with ABC. We see, from Figure 4.1 and Figure 4.3, that as the associativity of the A cache has increased, the NTS and PCS speedups have increased and the Victim scheme's speedup has dropped relative to the corresponding base cache, concurring with the results found in [6]. For the integer benchmarks, the performance of NTS is often on par with the best performing scheme,

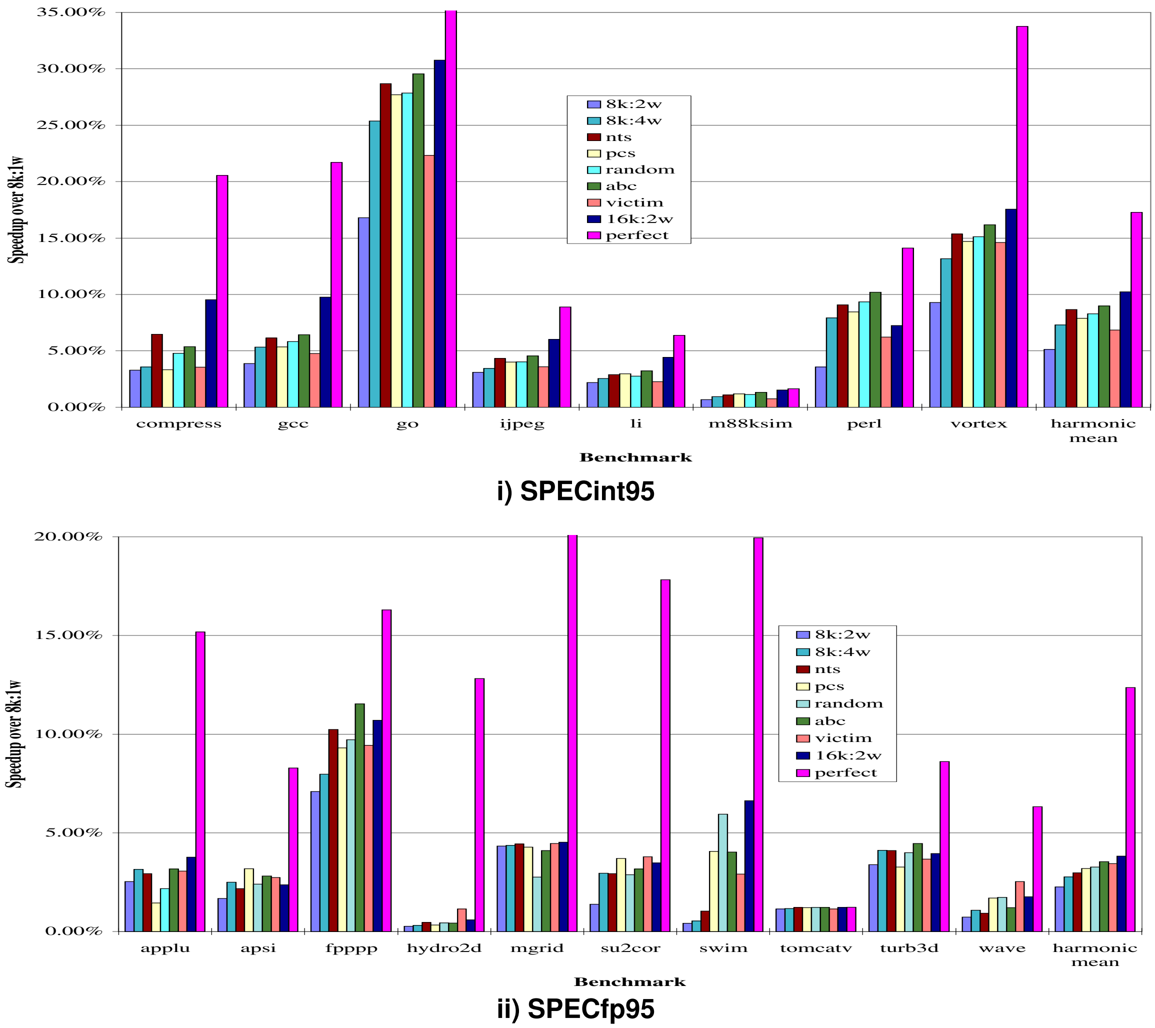
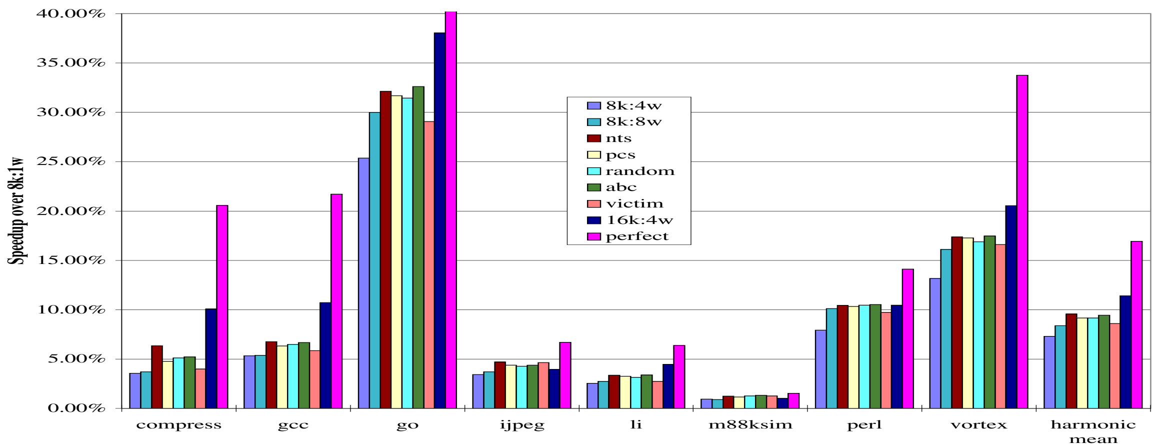


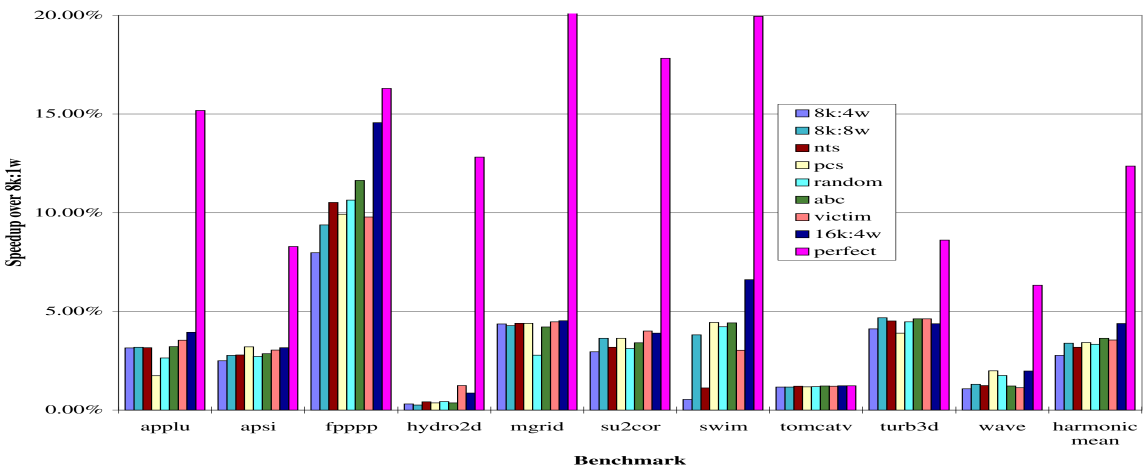
Figure 4.3: Overall execution speedup for the six evaluated cache schemes, relative to a single direct-mapped 8K cache (8k:1w). SPECint95 performance is shown on top, SPECfp95 below. Each of the multilateral schemes has a 2-way associative A cache. Perfect is the performance of a perfect cache. The harmonic mean speedup over the SPECint and SPECfp benchmarks is also shown.

ABC, and beats or equals PCS in every benchmark. In the floating point benchmarks, NTS' performance lags that of ABC somewhat, though it still beats PCS in over half of the floating point benchmarks (i.e. all but **apsi**, **su2cor**, **swim**, and **wave**).

The Victim scheme evaluated in these experiments incurs a 0 cycle latency for swaps, and yet its performance still trails that of ABC, and is often outperformed by NTS and PCS. If a nonzero swap latency is incurred (as it likely would be, in any real



i) SPECint95



ii) SPECfp95

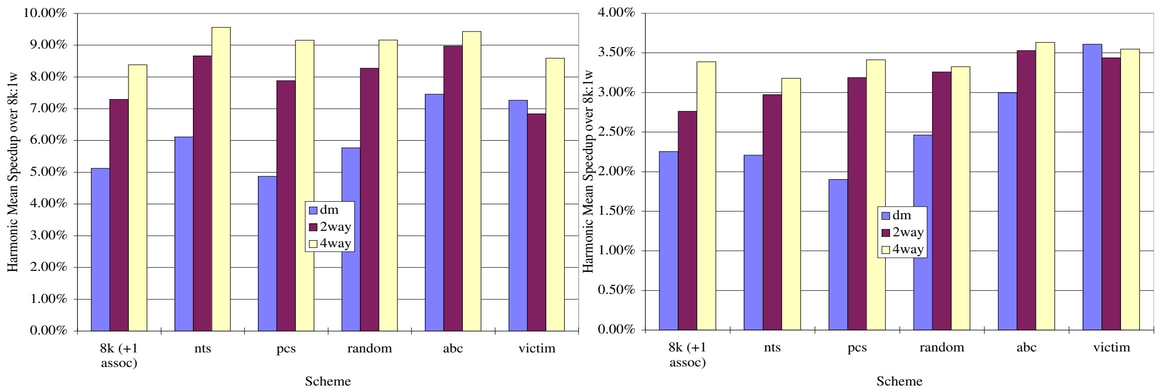
Figure 4.4: Overall execution speedup for the six evaluated cache schemes, relative to a single direct-mapped 8K cache (8k:1w). SPECint95 performance is shown on top, SPECfp95 below. Each of the multilateral schemes has a 4-way associative A cache. Perfect is the performance of a perfect cache. The harmonic mean speedup over the SPECint and SPECfp benchmarks is also shown.

---

implementation of Victim), Victim's performance would drop further.

The Random allocation scheme, interestingly, still performs quite competitively, though its overall performance is now exceeded by NTS in addition to ABC (except in harmonic mean SPECfp for 2-way multilateral structures, due to Random's high performance in **swim**). We shall discuss Random's performance further in Section 4.5.3.

As the associativity of the A cache is increased to 4-way, the performance of all of



i) SPECint95 **ii) SPECfp95**

Figure 4.5: Harmonic mean speedups of the multilateral cache schemes over 8k:1w as associativity increases. The SPECint95 speedups are shown on the left, SPECfp95 on the right. The effects of increasing associativity on a single structure cache are also shown under 8k (+1 assoc) – the *dm* bar indicates 8k:2w performance, *2way* shows 8k:4w performance, and *4way* shows 8k:8w performance.

---

the multilateral schemes begin to converge, with the NTS, PCS, and ABC schemes exhibiting comparable harmonic mean speedups over the base 8k:1w cache; another presentation of this trend is shown in Figure 4.5. All of these schemes outperform an 8-way cache of the same size (8k:8w) and are even competitive with a larger 4-way cache (16k:4w). Random's performance is comparable, though slightly lower than any of the "intelligent" multilateral cache schemes. Victim's speedup with 2-way or 4-way associative structures, even with a best case (0 cycle) swap latency, is beaten by all the multilateral schemes over the integer benchmarks and is outperformed by ABC over the floating point benchmarks.

4.4.3 The effect of a less aggressive processor engine

The experiments presented thus far have modeled a processor with perfect branch prediction and an aggressive out-of-order processor engine. However, imperfect branch prediction can greatly affect overall performance, as well as the performance improvement garnered by the use of better managed cache structures. Also, if a sufficient number of instruction buffer entries in the RUU is available to allow a large number of instructions to be in flight at once, more of the effects of cache misses may be masked by performing unrelated useful work.

We performed simulations of the same cache configurations (with direct-mapped A and fully-associative B caches) used in Section 4.3, except that in place of a perfect branch predictor, we used the Bi-Mode branch predictor [33], and we scaled the processor engine down to an effective 4-wide superscalar machine; a description of this less aggressive processor is shown in Table 4.3i. While other branch predictors may give slightly better performance for some of the benchmarks, the Bi-Mode branch predictor realistically represents the effect of one of today's best implementable branch prediction schemes on program execution times.

The IPCs attained by the perfect cache configuration coupled with this processor configuration are shown in Table 4.3ii, and the performance speedups of each of the cache schemes over the base 8K direct-mapped cache are shown in Table 4.4. We see that imperfect branch prediction and the reduced instruction fetch/issue/execution/retirement width has a large effect on overall performance, as expected: the IPC of the perfect cache configuration dropped from a low of 4.66 (**tomcatv**) and a high of 8.15 (**applu**) when the aggressive processor is used down to a low of 1.43 (**apsi**) and a high of 2.04 (**mgrid**) when the less aggressive processor is used.

The relative performance of the different cache schemes, however, still mirrors that shown in Figure 4.1: ABC has the best overall performance over the SPECint benchmarks, and Victim's performance is best overall over the SPECfp benchmarks, with ABC a close second. The ability of the processor to mask cache effects has been reduced due to the imperfect branch prediction and the smaller instruction buffers, which reduce the supply

Fetch Mechanism	fetches up to 4 instructions in program order per cycle
Instruction Cache	perfect cache, 1 cycle latency
Branch Predictor	Bi-Mode branch predictor
Issue Mechanism	out-of-order issue of up to 4 operations per cycle, 16 entry re-order buffer (RUU), 8 entry load/store queue (LSQ)
Functional Units	4 integer ALUs, 4 FP ALUs, 1 integer MULT/DIV, 1 FP MULT/DIV, 2 L/S units
Functional Unit Latency (total/issue)	integer ALU:1/1, integer MULT:3/1, integer DIV:12/12, FP adder:2/1, FP MULT:4/1, FP DIV:12/12. load/store:1/1
Data Cache	write-back, write-allocate, 32B lines, 2 read/write ports, non-blocking

i) less aggressive processor configuration

	Program	IPC		Program	IPC
SPECint95	**Compress**	1.8457	SPECfp95	**Applu**	1.5948
	Gcc	1.6032		**Apsi**	1.4255
	Go	1.4751		**Fpppp**	1.7343
	Ijpeg	1.9532		**Hydro2d**	1.6817
	Li	1.6672		**Mgrid**	2.0409
	M88ksim	1.9382		**Su2cor**	1.5528
	Perl	1.7261		**Swim**	1.5595
	Vortex	2.0307		**Tomcatv**	1.6129
				Turb3d	1.8920
				Wave5	1.4969

ii) IPCs for this processor configuration with perfect cache

Table 4.3: The less aggressive processor configuration (top) and the IPCs attained by this processor with a perfect cache configuration for the 18 benchmarks (bottom).

of useful work to do while waiting for cache delays; as a result, the best harmonic mean speedups, as seen by comparing Figure 4.1 to Table 4.4, have increased from 7.45% to 8.62% for ABC over SPECint and from 3.61% to 10.00% for Victim (2.99% to 9.57% for ABC) over SPECfp.

Even when we increase the cache miss (memory) latency by four times to 72 cycles, the performance ranking among the various schemes is very similar (though the absolute performance as measured by IPC decreases, as expected). ABC remains the best

	18 cycle miss latency					72 cycle miss latency				
	NTS	PCS	Victim	ABC	Rand	NTS	PCS	Victim	ABC	Rand
SPEC95 Integer Benchmarks										
Compress	3.27	2.13	3.49	**3.85**	3.12	10.32	7.09	11.48	**12.48**	10.32
Gcc	5.18	4.67	6.60	**6.86**	5.51	14.53	12.98	18.82	**19.45**	15.44
Go	13.42	11.96	**15.26**	14.75	12.62	30.47	26.99	**34.82**	33.70	28.74
Ijpeg	4.26	3.61	5.04	**5.05**	4.20	14.35	12.08	16.99	**17.11**	14.09
Li	5.31	4.02	5.33	**6.34**	5.10	15.06	11.25	15.43	**18.09**	14.53
M88ksim	4.51	3.84	5.35	**5.45**	4.89	18.75	16.73	21.85	**22.26**	20.07
Perl	6.83	6.39	7.90	**9.20**	7.90	22.15	20.24	**29.78**	29.77	25.73
Vortex	17.14	15.99	**20.16**	19.28	15.79	32.03	29.77	**37.84**	35.78	29.51
H-mean	*7.29*	*6.39*	*8.38*	***8.62***	*7.23*	*19.26*	*16.68*	*22.73*	***23.08***	*19.41*
SPEC95 Floating Point Benchmarks										
Applu	4.32	3.71	**4.78**	4.15	3.76	13.22	11.42	**14.58**	12.67	11.60
Apsi	3.40	5.97	**6.82**	6.17	5.00	12.91	20.67	**22.77**	20.82	16.80
Fpppp	15.99	7.93	15.43	**17.32**	12.90	34.92	15.88	31.98	**35.85**	26.68
Hydro2d	1.72	0.16	2.08	**2.49**	1.52	3.79	1.05	4.39	**5.45**	3.64
Mgrid	9.45	6.62	**9.80**	6.72	5.49	19.68	13.74	**20.37**	11.83	10.01
Su2cor	3.15	3.78	**4.48**	4.06	3.54	8.28	10.10	**11.64**	10.94	9.60
Swim	20.38	21.05	29.41	**30.37**	25.76	41.51	43.36	58.33	**59.20**	51.58
Tomcatv	6.71	5.86	**6.94**	6.90	6.76	26.86	26.66	**27.78**	27.66	27.03
Turb3d	3.70	3.06	**4.80**	4.16	4.05	9.64	8.28	**13.13**	11.22	11.10
Wave	15.44	13.90	**21.27**	19.57	18.66	34.30	29.82	**47.74**	44.04	42.06
H-mean	*8.08*	*6.91*	***10.00***	*9.57*	*8.27*	*19.27*	*17.00*	***23.39***	*21.95*	*19.35*

Table 4.4: Performance speedup (in %) over a single direct-mapped 8K cache for each scheme when a less aggressive processor (4-wide superscalar, with Bi-Mode branch prediction) is used and the miss latency is 18 cycles (left) and 72 cycles (right). H-Mean is the Harmonic mean speedup over the SPECint and SPECfp benchmarks. Best performing scheme for each benchmark (and Harmonic Mean) per miss latency is in bold. Victim has 0 cycle swap latency. Random (Rand) uses the 50/50 allocation ratio.

---

over SPECint, while Victim (with 0 cycle swap latency) remains best over SPECfp, with ABC close behind. Since the memory is four times slower than the other experiments, the effects of better cache management (i.e. higher hit rates) has a greater impact on overall performance speedup — the best harmonic mean speedups over the SPECint and SPECfp benchmarks are now 23.08% (ABC) and 23.39% (Victim), respectively.

	Compress	Gcc	Go	Li	Perl
opt	0.042	0.013	0.008	0.008	0.0169
pseudo-opt	0.048	0.016	0.023	0.010	0.0174
PONS	0.050	0.018	0.028	0.011	0.0175
NTS	0.063	0.029	0.054	0.018	0.0196
PCS	0.067	0.032	0.060	0.019	0.0190
Random	0.064	0.029	0.058	0.017	0.0207
Victim	0.062	0.025	0.047	0.016	0.0192
ABC	0.061	0.025	0.049	0.015	0.0199
16K:1w	0.063	0.029	0.050	0.020	0.0520

Table 4.5: Miss ratios for the eight (8+1)K and the 16K cache configurations on the trace inputs. *PONS* is the *pseudo-opt no-swap* scheme.

4.4.4 Near-optimal performance comparison

Table 4.5 shows the miss ratios experienced when running the five SPECint benchmark traces for an optimal scheme (*opt*), near-optimal scheme (*pseudo-opt*), *pseudo-opt* without swaps (*PONS*), NTS, PCS, ABC, Victim, and Random. The results among the implementable schemes mirror those found in the execution-driven simulations for an (8+1)K direct-mapped A, fully-associative B cache multilateral configuration (see Section 4.4.1) — except for **perl**, ABC and Victim are the best performing implementable schemes, and the performance difference between ABC and Victim is very small.

However, we notice that despite the improved performance attained by ABC over either NTS or PCS, there is still a large performance gap between ABC and a near-optimal management scheme with similar hardware constraints, *PONS*. Similarly, Victim trails its corresponding near-optimal scheme, *pseudo-opt*, which, like Victim, allows data movement between the caches. Table 4.6 shows the amount of the performance gap between NTS and *PONS* that is eliminated by Victim and ABC. We see that ABC eliminates up to 43% of the gap between the previous best overall proposed multilateral

	Compress	Gcc	Go	Li	Perl
Victim	7.69%	36.36%	26.92%	28.57%	19.05%
ABC	15.38%	36.36%	19.23%	42.86%	-14.29%

Table 4.6: Fraction of performance difference between NTS and *PONS* that ABC and Victim eliminate.

scheme, NTS, and *PONS*, the near-optimal management scheme that is closest in hardware configuration to ABC. Victim performs similarly, though ideally its comparison should be to *pseudo-opt*, as Victim allows movement between the caches. Nevertheless, we see that there is still a significant gap between the implementable multilateral schemes and the near-optimal schemes which benefit from perfect future knowledge. In **perl** (for this experiment), ABC has the highest miss ratio among all multilateral schemes, excluding Random; however, the miss ratio differences among all the schemes is very small. Moreover, the evaluation for a full program execution, as performed in Section 4.4.1, shows that ABC in fact has the highest speedup among all the implementable schemes for the (8+1)K cache configuration.

Thus, despite the advances in multilateral cache management schemes, they still have significant room for improvement. Possible avenues for addressing the remaining gaps to near-optimal performance will be presented in the discussion of future directions in Section 5.2. The next section is devoted to discussing the underlying causes of the performance differences we have seen.

4.5 Performance analysis

In this section, we use execution-driven simulation of the aggressive processor configuration to analyze the performance of the multilateral schemes running the SPEC95 benchmarks. We first compare and contrast the performance of the Victim scheme with the ABC scheme in Section 4.5.1. Then we discuss the performance differences between the ABC scheme and the NTS and PCS schemes in Section 4.5.2, and look at the Random scheme's performance in Section 4.5.3. We finish the analysis with a few notes about the ABC scheme's hardware requirements.

4.5.1 ABC vs. Victim

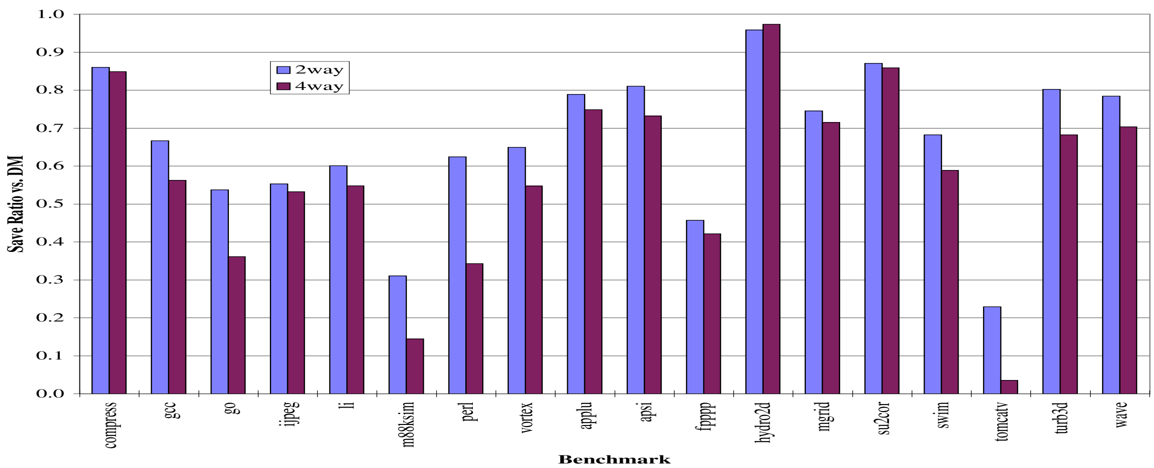
Inherently, the way both the ABC and Victim schemes attain their performance gains is similar: both reduce the impact of hot sets by providing dynamically-sized cache sets, and both achieve this by using the B cache as a buffer that may contain several blocks that map to the same set of the A cache. In both ABC and Victim, the allocation of some block to B occurs when a conflict occurs in cache A. However, beyond that, their approach to providing these dynamically-sized sets differs.

Victim in effect may create a larger size cache set whenever a conflict occurs in a set of A — the new miss block is allocated to A and the replaced block, the victim of that allocation, is saved in the B cache. If the element replaced from B due to the victim's allocation maps to the same set as the new miss block, the dynamic size of that set remains constant. If the replaced block and the victim are from different sets, the dynamic size of the victim's set increases, and that of the replaced block's set decreases by one.

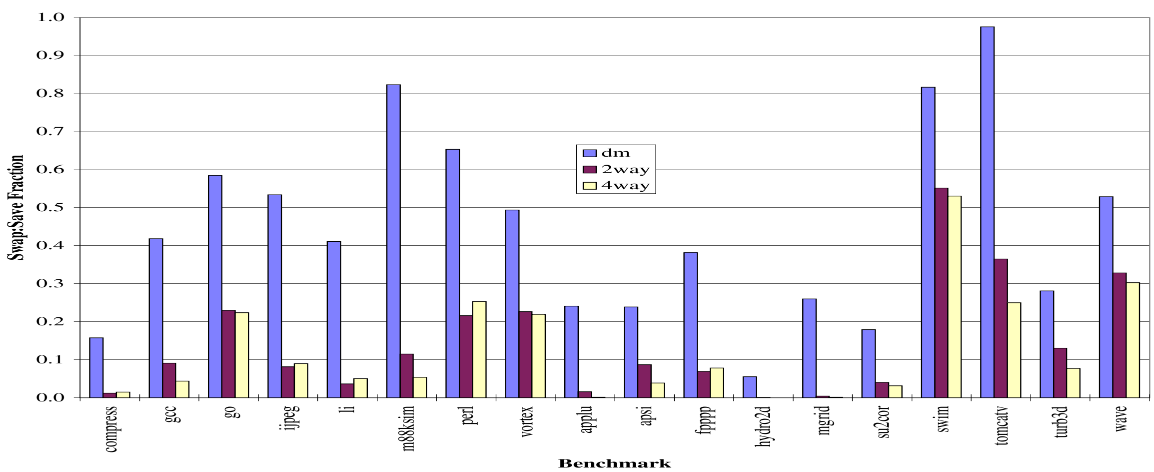
The ABC scheme, on the other hand, may only create a larger size cache set when a conflict is found in A and the conflict block has yet to experience a conflict, or the conflict block has been accessed since the last conflict to that A-set. In this case, the conflict block is retained in A and the miss block is allocated into B; if the miss block and the replaced block map to different sets in A, the dynamic size of the miss block's set increases by one block and the dynamic size of the replaced block's set decreases by one. In all other cases, the miss block is allocated into A and all set sizes remain the same.

Reducing the dynamic size of cache sets is beneficial when few of the conflicting elements are soon reused or need to reside in the cache concurrently. If, for instance, Victim saves an element α in the B cache that is not reused before it is evicted from B, α need not have been saved and some other element in B could instead have remained in L1. In the ABC cache, conflict elements in A may remain in A, but to do so, they must prove that they are useful by being accessed at least once between conflicts to their A-set. The B cache is then used to buffer miss blocks whose conflict blocks qualify to remain in A; blocks that are replaced from A or B return to the next level of memory.

When the A cache is direct-mapped, the ABC scheme outperforms the Victim cache in all but two of the SPECint95 benchmarks (**go** and **vortex**). In general, Victim is saving too many useless elements into the B cache and is not using the B cache as effectively as the ABC scheme. ABC's better usage of the B cache is due to its ability to allow elements to reside in B longer, as it is possible to evict items directly from A; in the Victim scheme, all evicted items from A must first travel through the B cache before they can leave the L1 cache structure. In the SPECfp95 benchmarks, however, this additional time in L1 for each evicted A cache element benefits Victim cache performance, which equals or beats ABC's performance in all the benchmarks except **swim**. Nevertheless, the difference in performance between Victim and ABC over the SPECfp benchmarks is less than 0.5%, and neither scheme has a speedup of more than 4% over the base cache. Note that these performance comparisons are for a Victim cache with 0 cycle swap latency; a realistic implementation of the Victim cache would incur at least a 1 or 2 cycle latency to exchange blocks between the caches for each swap. We have seen that as the swap latency



i) save ratio



ii) swap:save fraction

Figure 4.6: Victim cache save and swap characteristics as A cache associativity increases. The top figure shows the fraction of saves performed as A cache associativity increases relative to the number of saves performed for a direct-mapped A cache Victim configuration. The bottom figure shows the fraction of those saves performed that are eventually swapped back to A due to reaccess.

---

increases from 0 to 2 to 4 cycles, the Victim cache's performance degrades. Since no data movement at all is performed between the caches of the ABC scheme, swap latency is not an issue for ABC. Furthermore, ABC makes its performance gains without requiring the costly data path between the A and B caches that is needed by Victim.

As the associativity of the A cache increases, however, the Victim scheme's speedup is equaled or exceeded by the other multilateral schemes, as found in [6]. Victim's drop in relative performance occurs because the B cache in the Victim scheme is not as well utilized, as the sets in the A cache have less frequent conflicts and tend to generate fewer victims that benefit from being saved in the B cache. This phenomenon is well illustrated in Figure 4.6. The top of Figure 4.6 shows the decrease in the number of saves to the B cache by the Victim scheme as the associativity of A increases. This decrease occurs for two reasons: 1) as the associativity of A increases, the A cache typically is able to reduce the number of misses incurred, and thus fewer elements are replaced from A; 2) elements that are hit in B and swapped back to A have a higher likelihood of staying in A since the increased associativity of A can allow them to remain in the same set of A as 1 or 3 other useful blocks. Thus, even those blocks that are worth saving don't need to be saved as often.

The bottom portion of Figure 4.6 shows the fraction of blocks saved to B that are eventually swapped back to A. This ratio drops significantly as the associativity of A increases, indicating that fewer and fewer elements residing in B are ever actually accessed while in L1. Note that for 2-way and 4-way associative A caches the swap/save fraction is highest for **swim**, **tomcatv**, and **wave**, which may account for Victim's more competitive performance on SPECfp as the A cache associativity increases. Combining the effects of the two figures, we see that the absolute number of swaps being performed, and thus the number of useful elements found in the B cache of a Victim scheme, decreases greatly as A cache associativity increases. As a result, the performance benefit of the B cache in a Victim scheme degrades as the A cache associativity increases; less than 31% of the blocks placed in the B cache overall are actually reaccessed from the B cache when the A cache is 4-way associative.

The ABC scheme continues to perform well, however, since the B cache is used to hold data that is currently being referenced and found to conflict with useful data resident in the A cache; when the data in the conflicting A set is not found to be useful (i.e. it has not been accessed since the last conflict to that A-set), it is evicted from A and returns to the next level of memory; the B cache elements are thus allowed to remain there and potentially be reaccessed before their eviction.

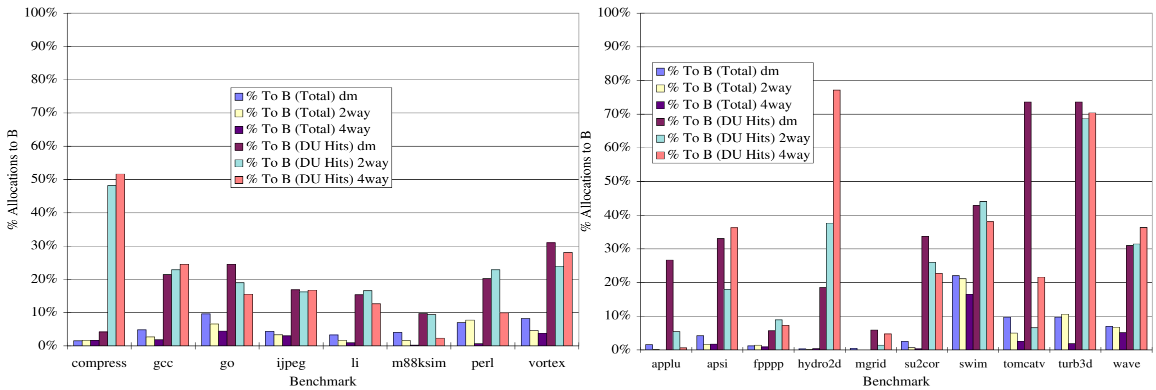
The performance of ABC pulls away from Victim in the SPECint benchmarks as the associativity of the A cache is increased, and ABC's performance exceeds Victim's performance in the SPECfp benchmarks. Note that the associative A cache results presented are for a Victim cache with 0 cycle swap latency; again, if a realistic swap latency is incurred, Victim's performance will degrade, as seen in the direct-mapped A cache performance numbers (see Figure 4.1).

4.5.2 ABC vs. NTS and PCS

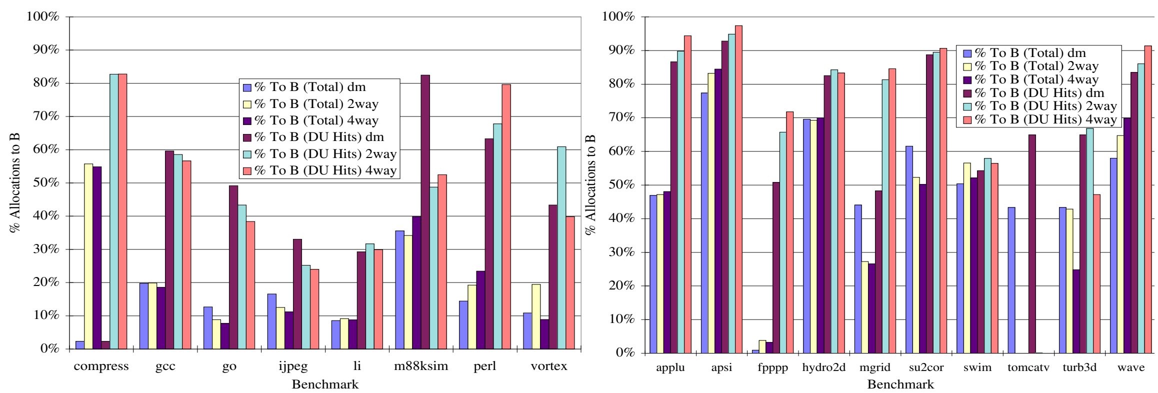
In previous studies, the NTS and PCS caches were found to perform best among the reuse-based multilateral allocation schemes [6][28][38] and to outperform Victim as the associativity of the A cache increased [6]. However, the ABC scheme outperforms both NTS and PCS when the associativity of the A cache is low and still performs comparably to the two as A cache associativity increases. This performance is especially interesting since the ABC scheme has much lower hardware requirements than either the NTS or the PCS schemes. While ABC requires only a single additional bit per block in cache A, the NTS and PCS schemes require bits for temporality detection for each block in A *and* B (typically one bit for each word in the block plus an additional temporality indicator bit), plus a Detection Unit for storing a tag and T/NT predictor; in these experiments, a 32-entry DU was used. The ABC scheme's allocation decision can be made using a simple logic circuit, while the NTS/PCS allocation decisions must be made via a table lookup with an associative compare. While these decisions are made only during miss handling and typically would not affect critical path latencies, the ABC scheme's implementation is decidedly simpler.

While the NTS and PCS schemes make their block allocation predictions based on past usage of miss blocks (NTS) and other blocks associated with them (PCS), the ABC scheme determines whether to allocate the new miss block into A or B by looking only at the *current* usage of the blocks *in cache A* that conflict with the miss block. Using the current state of the cache to make placement decisions can actually result in a better performance improvement than using past tour usage behavior, as tour usage behavior may not be persistent, as seen in [37][38]. Also, the ABC scheme makes its allocation decision based on whether to replace the conflict block in A, while the NTS/PCS schemes only look at the miss block's (NTS) or instruction's (PCS) temporality usage in previous tours. It is thus possible in NTS/PCS that many blocks are loaded into the A cache because they are marked T; if several of these blocks should indeed reside in the cache concurrently and the associativity of the A cache is not sufficient, blocks that will be accessed soon in the future will be kicked out. It would be better if all of these blocks could reside in the L1 at once, and the ABC scheme allows them to do so.

We can see the difference in allocation decisions by looking at how often each cache scheme allocates blocks to A or B. Recall that blocks are allocated to B in NTS and PCS only if the block or instruction, respectively, exhibited NT behavior during its most recent block tour and that information is still available in the DU. Figure 4.7 shows the fraction of tours that are allocated to the B cache in the NTS and PCS schemes for the SPEC95 benchmark suite. We show the fraction of B tours relative to both the total number of tours and also relative to the number of misses that hit in the DU. In general, for



i) NTS



ii) PCS

Figure 4.7: Fraction of tours that are allocated to the B cache in NTS (above) and PCS (below). SPECint95 benchmark results are shown on the left, SPECfp95 on the right. For each A cache associativity, the number of B cache tours is shown relative to the number of L1 tours (Total) and also to the number of cache misses that hit in the DU (DU Hits).

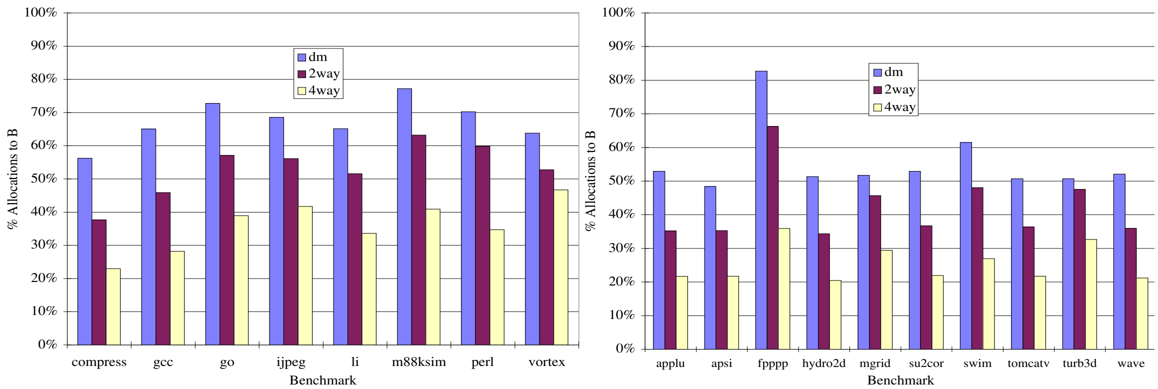
---

both NTS and PCS, we see that there is a large difference between these two fractions, indicating that many cache misses do not hit an entry in the DU and are allocated to the A cache by default. Another major point is that the B cache is not used very much in either NTS or PCS for the SPECint benchmarks — less than 23% of all block tours under PCS and less than 10% under NTS are allocated to B. While the size of the B cache is indeed much smaller than the A cache, we will see below that for the ABC and Random schemes, the B cache is used much more.

However, low usage of the B cache does not necessarily lead to low performance – – if the “correct” set of tours is allocated to B, then the overall performance of the scheme can be very good. For instance, data that is indeed streaming data (i.e. NTNS or NTS) would best be allocated to B regardless of the A cache state, as that data is unlikely to be reused very much, if at all.

Furthermore, the allocation decisions of NTS and PCS become more "correct" as the associativity of A increases. When the A cache is direct-mapped, many conflicts can occur and when a block is evicted, the block (NTS) or its loading instruction (PCS) is marked NT, causing future allocations of that block (NTS) or blocks loaded by that instruction (PCS) to go to the B cache. This occurs, of course, only when the reuse information is found in the DU. When the associativity of the A cache is low, too many conflicts occur and the DU is overrun by recently evicted block reuse information, so much so that little of it remains in the DU for use in future block allocation predictions. As the associativity of the A cache increases, fewer conflicts occur, the DU is not stressed as greatly, and blocks that are evicted from A that show nontemporal reuse are more likely to actually be nontemporal blocks; the longer a block stays in the cache, the more likely its reuse behavior is to reflect its optimal reuse characteristics [37][38]. As a result, more of the blocks that are allocated to B with a more highly associative A cache configuration are likely to actually be NT; placing these blocks in the smaller cache allows them to be accessed for a short while by the program while preventing them from evicting more useful, hopefully temporal, blocks that are resident in the large A cache. Thus, the performance of the NTS and PCS caches improves relative to Victim and Random as A cache associativity increases.

The ABC scheme more readily allocates blocks to B because its decisions are based on the current usage of blocks in the A cache at the time of the conflict. We see from Figure 4.8 that the fraction of blocks allocated to B decreases as the associativity of A



i) SPECint95 **ii) SPECfp95**

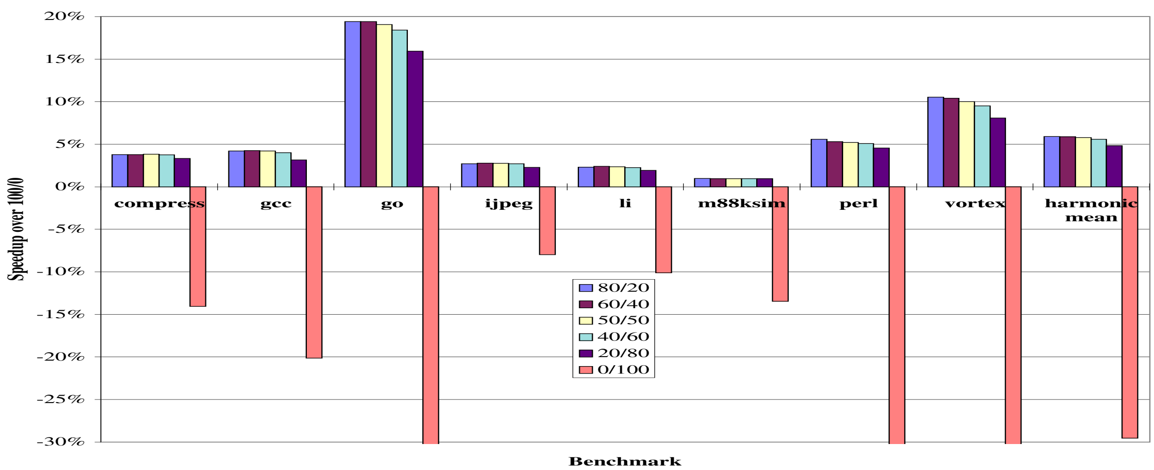
Figure 4.8: Fraction of tours that are allocated to the B cache in ABC. SPECint95 benchmark results are shown on the left, SPECfp95 on the right. For each A cache associativity, the number of B cache tours is shown relative to the total number of L1 tours.

increases. This phenomenon makes sense, as the number of conflicts typically decreases as the A cache associativity increases. Furthermore, as A cache associativity increases, the likelihood of finding a conflict block in the A-set that has not been accessed since the last conflict to that set increases as well, resulting in a potential increase in the number of blocks allocated to A. Nevertheless, even when the A cache is 4-way associative, the ABC cache still allocates at least 20% of all its tours to the B cache, a higher percentage than either NTS or PCS allocate to B (except for PCS in SPECfp). However, as the decision mechanisms for allocating to A vs. B differ between NTS, PCS, and ABC, allocating more blocks to B does not guarantee better performance.

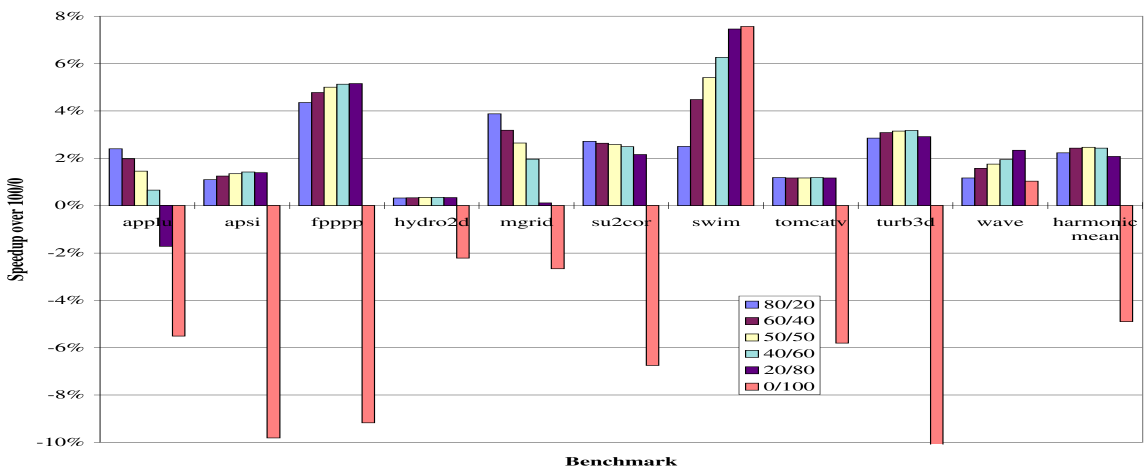
4.5.3 Random allocation performance

The Random allocation scheme is an interesting one, in that typically it would serve as a baseline for comparison for multilateral structures of a given size and configuration. In these experiments, we found that the Random scheme performed very well, actually better than either NTS or PCS in many cases when the A cache is direct-mapped. This performance is possible for several reasons: first, we have already evaluated some of the deficiencies seen for NTS and PCS when the A cache is direct-mapped, contributing to their sub-par performance. Second, allocating 50% of the tours to the B cache appears to be an acceptable ratio, similar to the fraction of tours that the ABC scheme allocates to B (see Figure 4.8). However, although the ABC scheme allocates approximately 50% of the tours to B for most of the benchmarks when a direct-mapped A cache is used, the ABC scheme's allocation decisions are directed by some reason; thus ABC should have a better chance than Random of picking a better set of particular tours to constitute the 50% allocated to B. Finally, the performance of Random is tied somewhat to the relative sizes and configurations of the A and B caches. When the ratio of A to B cache size is 8:1, as in these experiments, Random does quite well when the A cache is direct-mapped. However, as the associativity of A increases, Random's performance does not increase as much as NTS, PCS, or ABC. Perhaps some other allocation ratio would do better, particularly when the A and B cache associativities and the A to B size ratio differ from the cache configurations evaluated here.

We evaluated the effects of varying the allocation ratio from 50/50, with one extreme being 100/0 (all allocations made to the A cache, reducing to the single direct-mapped 8K cache performance) and the other extreme being 0/100 (all allocations made to the B cache, reducing to the performance of a single 1K fully-associative cache). The speedup attained by each configuration is shown in Figure 4.9. We see that, in general, for the SPECint benchmarks, as the allocation ratio increases (allocating more blocks to B), performance slowly degrades, with the performance of 0/100 (allocating all blocks to B) consistently resulting in a severe decrease in performance relative to the base cache. However, the trends for the SPECfp benchmarks are not as consistent. In some benchmarks, such as **applu** and **su2cor**, performance degrades as more blocks are



i) SPECint95



ii) SPECfp95

Figure 4.9: Speedups over a single 8K direct-mapped cache as the allocation ratio for Random is varied from 80/20 to 0/100. 100/0 is identical to the single 8K direct-mapped cache, and thus is the baseline. 0/100 speedup for go is -62%, perl -33%, vortex -30%, and turb3d -13%.

---

allocated to B, as seen for the SPECint benchmarks. However, in workloads like **apsi** and **fpppp**, performance increases as more blocks are allocated to B, until only the B cache is used, at which point the performance drops severely. Interestingly, for **swim** and **wave**, the performance of a 1K fully-associative cache (0/100) is higher than that of the baseline 8K direct-mapped cache. This is likely due to poor data layout resulting in very high contention for a small number of sets. Over all the benchmarks, in terms of harmonic mean speedup, the 50/50 ratio is the most consistent performer. If certain workloads are

	NTS	PCS	Victim	ABC	Random					
					80/20	60/40	50/50	40/60	20/80	0/100
SPEC95 Integer Benchmarks										
Compress	5.06	0.52	3.86	**5.12**	3.77	3.77	3.81	3.75	3.32	-14.06
Gcc	4.28	3.50	5.11	**5.31**	4.19	4.25	4.19	3.99	3.14	-20.15
Go	20.93	19.21	**22.48**	21.86	19.40	19.40	19.04	18.40	15.92	-62.45
Ijpeg	2.76	2.08	3.75	**3.81**	2.69	2.76	2.75	2.68	2.27	-8.00
Li	1.97	1.45	2.03	**2.47**	2.30	2.39	2.35	2.24	1.93	-10.11
M88ksim	0.47	0.90	0.81	0.94	**0.96**	0.94	0.95	0.94	0.93	-32.64
Perl	5.05	3.47	8.95	**9.68**	5.58	5.29	5.18	5.07	4.55	-32.64
Vortex	11.03	10.41	**14.43**	13.37	10.52	10.40	10.01	9.50	8.08	-30.34
H-Mean	*5.12*	*6.11*	*7.26*	***7.45***	*5.89*	*5.87*	*5.77*	*5.57*	*4.82*	*-29.54*
SPEC95 Floating Point Benchmarks										
Applu	2.17	0.82	**3.22**	2.01	2.40	1.97	1.46	0.65	-1.73	-5.51
Apsi	1.46	0.95	**2.80**	2.35	1.09	1.25	1.35	1.41	1.39	-9.82
Fpppp	6.38	4.05	**7.25**	6.46	4.36	4.78	5.00	5.12	5.16	-9.18
Hydro2d	0.15	0.08	**1.29**	0.47	0.32	0.33	0.34	0.35	0.34	-2.22
Mgrid	3.27	2.54	**4.30**	3.83	3.87	3.19	2.64	1.96	0.11	-2.67
Su2cor	2.58	3.06	**4.12**	2.91	2.72	2.64	2.58	2.49	2.16	-6.76
Swim	0.93	2.39	5.16	5.47	2.50	4.48	5.41	6.26	7.46	**7.56**
Tomcatv	1.20	1.20	1.20	**1.21**	1.18	1.17	1.17	1.18	1.16	-5.81
Turb3d	3.40	2.81	**3.98**	3.80	2.85	3.08	3.15	3.18	2.91	-12.57
Wave	0.80	1.23	**3.05**	1.75	1.16	1.57	1.76	1.94	2.34	1.03
H-Mean	*2.21*	*1.90*	***3.61***	*2.99*	*2.23*	*2.43*	*2.46*	*2.42*	*2.07*	*-4.91*

Table 4.7: Performance speedup (in %) over a single direct-mapped 8K cache for each (8+1)K direct-mapped A, fully-associative B cache scheme. H-Mean is the Harmonic mean speedup over the SPECint and SPECfp benchmarks. Best performing scheme for each benchmark (and Harmonic Mean) is in bold. Victim has 0 cycle swap latency.

targeted, the ratio of allocations in the Random scheme might be set to attain higher performance, but the 50/50 ratio appears to be best overall.

A summary of the performance speedups for the (8+1)K direct-mapped A, fully-associative B configurations is shown in Table 4.7 while the performance speedups for the (8+1)K 2-way A and (8+1)K 4-way A cache configurations are shown in Table 4.8. While the Random scheme's performance is reasonable, its behavior is not easily predictable

	2-way A cache					4-way A cache				
	NTS	PCS	Victim	ABC	Rand	NTS	PCS	Victim	ABC	Rand
SPEC95 Integer Benchmarks										
Compress	**6.45**	3.32	3.55	5.36	4.78	**6.34**	4.75	3.99	5.21	5.11
Gcc	6.13	5.35	4.75	**6.41**	5.80	**6.74**	6.32	5.85	6.67	6.48
Go	28.68	27.69	22.32	**29.56**	27.84	32.13	31.67	29.04	**32.59**	31.44
Ijpeg	4.32	4.00	3.57	**4.54**	4.02	**4.69**	4.38	4.63	4.37	4.25
Li	2.88	2.95	2.26	**3.22**	2.74	3.36	3.25	2.72	**3.38**	3.14
M88ksim	1.07	1.17	0.73	**1.30**	1.12	1.23	1.14	1.25	**1.33**	1.25
Perl	9.08	8.45	6.22	**10.18**	9.33	10.43	10.32	9.71	**10.52**	10.45
Vortex	15.34	14.70	14.60	**16.17**	15.11	17.38	17.26	16.60	**17.46**	16.88
H-mean	*8.66*	*7.88*	*6.84*	***8.97***	*8.27*	***9.56***	*9.15*	*8.59*	*9.43*	*9.16*
SPEC95 Floating Point Benchmarks										
Applu	2.93	1.45	3.06	**3.17**	2.17	3.15	1.73	**3.54**	3.21	2.63
Apsi	2.17	**3.17**	2.73	2.80	2.39	2.78	**3.20**	3.03	2.85	2.71
Fpppp	10.23	9.30	9.43	**11.53**	9.71	10.52	9.92	9.78	**11.63**	10.64
Hydro2d	0.45	0.33	**1.14**	0.42	0.44	0.41	0.35	**1.23**	0.36	0.43
Mgrid	4.43	4.27	**4.46**	4.10	2.75	4.39	4.39	**4.46**	4.21	2.78
Su2cor	2.93	3.70	**3.78**	3.17	2.87	3.18	3.63	**4.00**	3.40	3.11
Swim	1.03	4.05	2.90	4.02	**5.94**	1.12	**4.44**	3.02	4.41	4.22
Tomcatv	**1.21**	1.20	1.14	**1.21**	**1.21**	1.20	1.17	1.20	**1.21**	1.18
Turb3d	4.09	3.26	3.66	**4.46**	3.99	4.50	3.89	**4.62**	**4.62**	4.47
Wave	0.93	1.69	**2.53**	1.21	1.21	1.24	**1.99**	1.13	1.22	1.75
H-mean	*2.97*	*3.19*	*3.44*	***3.53***	*3.26*	*3.18*	*3.41*	*3.55*	***3.63***	*3.32*

Table 4.8: Performance speedup (in %) over a single direct-mapped 8K cache for each (8+1)K 2-way (left) and 4-way (right) A, fully-associative B cache scheme. H-Mean is the Harmonic mean speedup over the SPECint and SPECfp benchmarks. Best performing scheme for each benchmark (and Harmonic Mean) per A cache associativity is in bold. Victim has 0 cycle swap latency. Random (Rand) uses the 50/50 allocation ratio.

across different benchmarks. Schemes whose allocation decisions are based on some current or past reuse information, like the ABC, NTS, and PCS schemes, do indeed perform better overall as the cache configurations are varied.

4.5.4 Alternative hardware implementations for ABC

The ABC scheme has the lowest hardware requirements of any proposed multilateral scheme, yet performs best overall for the evaluated SPEC95 benchmarks. ABC makes its performance gains without the use of a costly data path between the caches, as required by Victim, and without the temporality detectors and Detection Unit required by NTS and PCS. ABC only requires the addition of a single bit per block in A and a very simple logic circuit for making its management decisions.

Although the ABC scheme described above requires only a single additional bit (the Conflict bit) per block in A for making its allocation decisions, the size of the Conflict detector can be increased from the 1-bit scheme presented to multiple-bit saturating counters. For example, the Conflict status could be kept via a 2-bit counter that would be incremented each time a conflict to that set occurred and reset each time the block is referenced. Allocation of a miss block to A would be done only when the conflict block has a count of 3. An increase in the Conflict counter size would force more blocks to be allocated to B, as an incoming miss block could be considered for A cache allocation only after the Conflict counter has reached its maximum value. For the cache configurations and benchmarks evaluated, we found that the best performance was attained by using a single bit for the Conflict detector.

For an associative A cache, the ABC scheme sets the Conflict bit of all the blocks in the conflicting A-set to 1 when a conflict is found. Alternatively, the C bit could be set only for the LRU element of the conflicting A set, as that is the block that would have been evicted were the new miss block to be allocated to A. While this scheme would require only one Conflict bit per set of A (associated with the LRU block), as opposed to a

Conflict bit per block in A, this scheme would allocate more blocks to B than the original ABC scheme, since the LRU element of A can be replaced only if a conflict to this set has occurred since this element became LRU. For associative caches, the time since becoming LRU is less than the time since the last reference to the conflict block, used above, and hence the conflict block is less likely to have its `C=1`, and thus is less likely to be replaced on conflict. We found in these experiments that while this LRU-marking scheme sometimes performed well, the entire-set marking scheme of the ABC scheme used above performed best overall.

4.6 Summary

In this chapter we have introduced a new multilateral cache allocation scheme, Allocation By Conflict (ABC), and compared its performance to two previously proposed reuse-based multilateral schemes, NTS and PCS, a third multilateral scheme, the Victim cache, and a multilateral performance baseline, Random. As shown in earlier work, the performance of each of the multilateral schemes is better than a more associative single structure cache of comparable size (i.e. direct-mapped A, fully-associative B cache multilateral configurations outperform 2-way single structure caches; 2-way A, fully-associative B multilateral configurations outperform 4-way single structure caches, etc.) and is comparable to, or even better than traditional caches with the same A cache associativity of nearly twice the size. We have shown that the ABC scheme performs best over all of the reuse-based allocation schemes because it makes its allocation decisions based on the current usage of blocks in cache vs. the past tour usage of incoming miss blocks, as done in NTS and PCS. Furthermore, the ABC scheme outperforms the Victim scheme over the SPECint95 suite, and performs comparably over the SPECfp95 benchmarks when the associativity of the A cache is low. The Random scheme also performs quite well for these cache configurations and sizes; however, its performance is less predictable and is likely to change as the cache configurations are varied. As the associativity of the A cache increases, the performance of Random and Victim degrade relative to NTS, PCS, and ABC.

The ABC scheme has the lowest hardware requirements of any of the evaluated multilateral cache schemes, requiring only a single additional bit per block in the A cache. The NTS and PCS schemes require additional temporality detection bits per block and a Detection Unit for storing reuse information, while the Victim scheme requires a costly data path between the caches for saves and swaps. NTS and PCS do perform comparably to ABC as the associativity of the A cache increases. However, their implementation complexity is much higher than ABC's, and ABC is the best performing cache scheme overall.

CHAPTER V

CONCLUSIONS AND FUTURE DIRECTIONS

5.1 Conclusions

Minimizing the *average data access time* is of paramount importance when designing high-performance machines. Unfortunately, access time to off-chip memory (measured in processor clock cycles) has increased dramatically as the disparity between main memory access times and processor clock speeds widens. The effect of this disparity is further compounded as multiple-issue processors continue to increase the number of instructions that can be issued each cycle.

While there are many approaches to minimizing the average data access time, the most common solutions are to incorporate multiple levels of cache memory on-chip and increase the capacity of on-chip caches, but still allocate and replace their blocks in a manner that is essentially the same as when caches first appeared three decades ago. *Active cache management* can improve the performance for a given size cache. Active management uses information, such as a block's reference behavior, to make block allocation and/or replacement decisions; this is opposed to traditional, more passive cache management in which blocks are allocated into a cache structure via a simple modulo mapping of their addresses and replaced according to policies, like LRU, which exploit only a very limited amount and type of behavioral information.

In order to accommodate varying block usage within a given cache structure, several schemes [3][4][5][6] have been proposed that incorporate an additional data store within the L1 cache structure to manage the state of the resulting *multilateral* [7] cache by exploiting reuse pattern information. The previously proposed schemes perform active block allocation, though they do not perform active replacement. Each scheme instead relegates block replacement decisions to each data store, which uses its own simple hardware replacement algorithm.

Our approach to improving cache performance, and thus reducing average data access time via active cache management, has resulted in two main contributions. The first is the development and deployment of a flexible, timing-sensitive multilateral cache simulator, *mlcache*. *mlcache* allows one to easily model and compare the performance of many different multilateral cache designs on a common platform, allowing one to control the cache design parameters so as to permit a fair comparison of each scheme's performance based on the management decisions it makes. *mlcache* also allows for in-depth analysis of each scheme's performance via the detailed outputs the simulator provides; beyond simple hit/miss ratios, delayed hit ratios, average access latencies, and reuse behavior, among others, are useful metrics that are provided to help identify strong and weak points of an evaluated cache scheme. Using *mlcache*, we have been able to evaluate the performance of proposed multilateral cache schemes, and to derive new, better performing schemes.

The second contribution of this work is the introduction of a new multilateral allocation scheme, Allocation By Conflict (ABC), which in our evaluations outperforms all previously proposed reuse-based multilateral configurations and performs comparably to multilateral schemes with much higher hardware requirements (e.g. Victim [18], which requires a data path between the A and B caches). The ABC scheme incurs the lowest hardware cost of any of the proposed multilateral schemes, yet it performs the highest and is the most easily implementable. The ABC scheme requires one additional bit per block in the cache A directory, and a very simple logic circuit for making the allocation decisions. The ABC scheme's performance also scales well as the sizes of the caches are increased and also, in contrast to the Victim cache, as the associativity of the A cache is increased.

5.2 Future directions

There are many different directions that research on active cache management can take in the future. Two areas which may have a direct impact on overall cache and memory system performance are the incorporation of more active replacement policies and the use of compiler/software hints. These directions can be employed individually or in conjunction; likely the best performing active cache management scheme will include some facet of each.

5.2.1 Active replacement

Throughout this dissertation, active allocation was the only part of active cache management addressed; replacement decisions were still relegated to each cache's hardware algorithm (generally LRU). Although ABC in effect does use active replacement in making its allocation decisions (by deciding whether to replace the LRU block in the A-set), it still ends up replacing either the LRU block in the A-set or the LRU block in the B-set when a miss occurs. Though we have made significant gains in cache performance via active allocation, making better decisions for block replacement can further improve performance for a given cache size. For example, often the LRU element λ is not the optimal replacement choice: a block γ that is more recently used might be a better replacement candidate than λ, as λ may be used sooner in the future than block γ. If γ is replaced instead of λ, λ may be able to remain in cache for its next reference. By exploiting usage information, we may be able to detect these better eviction candidates and actively replace them from the cache when replacements must be made. If done successfully, more useful blocks will be allowed to remain in cache for longer periods of time, likely improving program performance further.

5.2.2 Compiler/software hints

Using hardware-only based mechanisms, the decision-making engines, whether making allocation or replacement decisions, must always endure a period of learning before a good decision can be made. If the decisions arrived at from these learning processes are long standing and persistent, then the penalty for learning is low. However, if the best decisions change frequently, the learning process may continue indefinitely, and implementable schemes will often be forced to employ some lower-performing default allocation or replacement mechanism.

When learning curves are long and the payoffs are small, having hints from the compiler or the programmer can help each scheme perform better than their default decisions would effect. For instance, if it is known by the compiler or programmer that loaded data will be accessed in a nontemporal fashion (i.e. streaming data), then a hint could be sent to the hardware to place the data in the smaller B cache of the multilateral cache structure. This decision would allow such blocks to remain in L1 for a short period of time, as needed, while preventing them from wreaking havoc on the larger, less associative A cache. While the nontemporal nature of the block might be detected via hardware, its initial placement would already have been made, and the performance degrading effect of a poor placement could have been avoided if a hint were provided.

Thus, the incorporation of compiler or software hints can help alleviate the poor decisions that might be made when no hardware-gathered information for a block is available, or when the decisions made for that block are not persistent. The hints can thus help make better allocation and/or replacement decisions for a particular block, thereby enabling the hardware to make even better decisions about other blocks which it does track well. Unfortunately, the incorporation of compiler or software hints would require target programs to be recompiled, so the improvement gained by having the hints must be weighed against the cost of reworking the code.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture*, *A Quantitative Approach*, Morgan Kaufman Publishers Inc., San Mateo, CA, 1990.

[2] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A Modified Approach to Data Cache Management," *Proceedings of MICRO-28*, pp. 93–103, December 1995.

[3] J. A. Rivers and E. S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," *Proceedings of the 1996 ICPP*, vol. I, pp. 151–160, August 1996.

[4] T. Johnson and W. W. Hwu, "Run-time Adaptive Cache Hierarchy Management via Reference Analysis," *Proceedings of ISCA-24*, pp. 315–326, June 1997.

[5] A. Gonzalez, C. Aliagas, and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *Proceedings of the 1995 Conference on Supercomputing*, pp. 338–347, July 1995.

[6] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens, "Utilizing Reuse Information in Data Cache Management," *Proceedings of the 1998 ICS,* pp. 449–456, July 1998.

[7] J. A. Rivers, E. S. Tam, and E. S. Davidson, "On Effective Data Supply for Multi-Issue Processors," *Proceedings of the 1997 ICCD*, pp. 519–528, October 1997.

[8] G. S. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proceedings of ASPLOS-IV*, pp. 53–62, April 1991.

[9] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proceedings of ISCA-8*, pp. 81–85, May 1981.

[10] M. J. Charney and T. R. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark suite," *IBM Journal of Research and Development*, Vol. 41 Number 3, pp. 265–286, May 1997.

[11] J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proceedings of Supercomputing'91*, pp. 176–186, November 1991.

[12] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proceedings of ASPLOS-IV*, pp. 40–52, April 1991.

[13] T.-F. Chen and J.-L. Baer, "Reducing Memory Latency via Non-Blocking and Prefetching Caches," *Proceedings of ASPLOS-V*, pp. 51–61, April 1992.

[14] V. Milutinovic, A. Milenkovic, and G. Sheaffer, "The Cache Injection/Cofetch Architecture: Initial Performance Analysis," *Proceedings of MASCOTS'97*, Haifa, Israel, pp. 63–64, January 1997.

[15] A. Seznec, "A Case for Two-way Skewed-associative Caches," *Proceedings of ISCA-20*, pp. 169–178, May 1993.

[16] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating Cache Conflict Misses through XOR-based Placement Functions" *Proceedings of the 1997 ICS*, pp. 76–83, July 1997.

[17] A. Agarwal and S. D. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," *Proceedings of ISCA-20*, pp. 179–190, May 1993.

[18] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," *Proceedings of ISCA-17*, pp. 364–373, June 1990.

[19] E. S. Tam and E. S. Davidson, "Early Design Cycle Timing Simulation of Caches," *Technical Report CSE-TR-317-96,* University of Michigan, November 1996.

[20] J-D Wellman and E. S. Davidson, "The Resource Conflict Methodology for Early-Stage Design Space Exploration of Superscalar RISC Processors," *Proceedings of the 1995 ICCD*, pp. 110–115, October 1995.

[21] D. Burger and T. M. Austin, "Evaluating Future Microprocessors: the SimpleScalar Tool Set," *Technical Report #1342*, University of Wisconsin, June 1997.

[22] G. Kurpanchek et al., "PA-7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface." *COMPCON Digest of Papers*, pp. 375–382, February 1994.

[23] E. Rashid et al., "A CMOS RISC CPU with On-Chip Parallel Cache," *ISSCC Digest of Papers*, pp. 210–211, February 1994.

[24] M. D. Hill, DineroIII Documentation, Unpublished UNIX-style Man Page, University of California, Berkeley, October 1985.

[25] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin, "On High-Bandwidth Data Cache Design for Multi-Issue Processors," *Proceedings of MICRO-30*, pp 46–57, December 1997.

[26] T-P Shih, "Goal-Directed Performance Tuning for Scientific Applications," *Ph.D. Dissertation*, University of Michigan, Ann Arbor, Michigan, 1996.

[27] J-L. Baer and W-H. Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proceedings of ISCA-15*, pp. 73–80, May 1988.

[28] E. S. Tam, J. A. Rivers, and E. S. Davidson, "Flexible Timing Simulation of Multiple Cache Configurations," *Technical Report CSE-TR-348-97,* University of Michigan, November 1997.

[29] R. C. Bedicheck, "Talisman: Fast and Accurate Multicomputer Simulation," *Proceedings of the 1995 ACM SIGMETRICS Conference*, pp. 14–24, May 1995.

[30] P. Magnusson and B. Werner, "Efficient Memory Simulation in SimICS," *Proceedings of the 28th Annual Simulation Symposium*, pp. 62–73, April 1995.

[31] J. Reilly, "SPEC Describes SPEC95 Products and Benchmarks," *SPEC Newsletter*, September 1995.

[32] E. S. Tam, J. A. Rivers, G. S. Tyson, and E. S. Davidson, "*mlcache:* A Flexible multilateral Cache Simulator," *Proceedings of MASCOTS'98*, pp. 19–26, July 1998.

[33] C-H. Chi and H. Deitz, "Improving Cache Performance by Selective Cache Bypass," *Proceedings of 22nd Hawaii International Conference on System Science*, pp. 277–285, January 1989.

[34] C-C. Lee, I-C. K. Chen, and T. N. Mudge, "The Bi-Mode Branch Predictor," *Proceedings of MICRO-30*, pp. 4–13, December 1997.

[35] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Systems Journal, Vol. 5,* pp. 78–101, 1966.

[36] V. Srinivasan and E. S. Davidson, “Improving Performance of an L1 Cache with an Associated Buffer,” *Technical Report CSE-TR-361-98*, University of Michigan, March 1998.

[37] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. S. Davidson, “Evaluating the Performance of Active Cache Management Schemes,” *Proceedings of the 1998 ICCD*, pp. 368-375, October 1998.

[38] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. S. Davidson, “Active Management of Data Caches by Exploiting Reuse Information,” *To Appear in IEEE Transactions on Computers*, 1999.